Mise en oeuvre d'un décodeur MPEG-2 sur architecture mixte processeur + FPGA

Claire Billaud
Élève en 3ème année IS
Mars-Juillet 2004

ENSEA - Laboratoire ETIS
Équipe Architecture

Responsable administratif : M. Didier DEMIGNY *(demigny@ensea.fr)*
Responsable technique : M. François VERDIER *(verdier@ensea.fr)*

Résumé

Cette étude vise à implémenter un décodeur vidéo MPEG-2 dans une architecture mixte composée d'un microprocesseur et d'un FPGA. Elle part d'un décodeur MPEG-2 purement logiciel dont on fait l'analyse pour en extraire des fonctions intéressantes à implémenter en FPGA : la déquantification, la DCT inverse et la prédiction de mouvement. Ces fonctions, une fois identifiées, font l'objet d'une implementation hardware. La configuration de la plate-forme cible, l'adaptation du programme à cette plate-forme et l'interfaçage entre le FPGA et le microprocesseur sont également pris en considération.

Abstract

This study aims to implement an MPEG-2 video decoder in an mixed architecture composed of a microprocessor and a FPGA. It starts with a MPEG-2 decoder software which is analyzed in order to extract functions that can be interestingly implemented in an FPGA : dequantization, inverse DCT and motion prediction. Once identified, these functions are implemented in hardware. Configuring the target platform, adapting the program to that platform and interfacing between the FPGA and the microprocessor is also considered.

Mots-clés

Compression/décompression vidéo MPEG-2
Architecture mixte
Processeur Power-PC
FPGA

Keywords

MPEG-2 video compressing/uncompressing
Mixed architecture
Power-PC processor
FPGA

Remerciements

Je tiens à remercier ici l'ENSEA et l'équipe de recherche ETIS pour leur accueil, en particulier M. François Verdier pour l'aide qu'il m'a apportée tout au long de mon stage.
Je remercie Arthur Segard pour m'avoir grandement aidée à la prise en main de la plate-forme et de ses outils de développement, et pour avoir résolu certains problèmes inattendus.
Je remercie M. Pascal Téboul, le professeur chargé du suivi de mon stage, pour sa disponibilité.
Je remercie tous ceux qui, de près ou de loin, m'ont soutenue et ont ainsi contribué au bon déroulement de ce projet.

Table des matières

1 - L'ENSEA et le laboratoire de recherche ETIS **7**

1.1 - L'ENSEA **7**

1.2 - Le laboratoire de recherche ETIS **7**

1.3 - L'équipe Architecture **8**

2 - Mise en oeuvre du décodeur MPEG-2 **9**

2.0 - Introduction générale **9**

2.1 - Le format MPEG-2 **10**
2.1.1 - La compression d'images JPEG **10**
2.1.2 - La compression vidéo MPEG **12**
2.1.3 - Le format MPEG-2 **14**

2.2 - Le décodeur MPEG-2 logiciel **17**
2.2.1 - Présentation générale **17**
2.2.2 - Fichiers et fonctions du décodeur **18**
2.2.3 - Recherche des fonctions susceptibles d'être implémentées en hardware **20**

2.3 - Analyse des fonctions à implémenter en hardware **22**
2.3.1 - Analyse de la fonction d'IDCT **22**
2.3.2 - Analyse de la fonction de déquantification **22**
2.3.3 - Performances "brutes" de l'IDCT et de la déquantification **23**
2.3.4 - Parallélisation de l'IDCT **24**
2.3.5 - La fonction de prédiction de mouvement **25**
2.3.6 - Analyse de `form_component_prediction` **26**
2.3.7 - Parallélisation de la prédiction de mouvement **27**

2.4 - Conception des fonctions hardware **29**
2.4.1 - Conception de l'ensemble IQ+IDCT **29**
2.4.2 - Conception de la déquantification **29**
2.4.3 - Conception de l'IDCT **33**

3 - Résultats de l'implémentation sur la plate-forme **35**

3.1 - La plate-forme mixte **35**
3.1.1 - Présentation **35**
3.1.2 - Programmation de la plate-forme mixte **36**

3.2 - Gestion de la mémoire **38**

3.2.1 - Chargement d'un programme dans la SDRAM **38**
3.2.2 - Le linker script **38**
3.2.3 - Montage de la SDRAM **39**

3.3 - Adaptation du système de fichiers **41**
3.3.1 - Le Memory File System (MFS) **41**
3.3.2 - Utilisation du MFS **41**
3.3.3 - Les limites du MFS **42**
3.3.4 - Transfert des fichiers **42**

3.4 - Résultats de l'implémentation **44**
3.4.1 - Vérification des résultats **44**
3.4.2 - Les futures améliorations **44**

3.5 - Conclusion **46**

Bibliographie **47**

Annexe : Xilinx Embedded Development Kit (EDK) **48**
Annexe : Xilinx ISE Project Navigator **52**
Annexe : XMD et la console *xygwin* **54**
Annexe : le protocole *Xmodem* **56**

Table des figures

Figure 1 : Répartition des éléments de luminance et de chrominance selon les normes 10
Figure 2 : Quantification d'un bloc 11
Figure 3 : Réarrangement des images dans un flux MPEG 13
Figure 4 : Schéma simplifié d'un décodeur MPEG ou MPEG-2 14
Figure 5 : Combinaison des profils et des niveaux en MPEG-2 15
Figure 6 : Découpage des blocs en mode trame 16
Figure 7 : Graphe d'appel des fonctions du décodeur 19
Figure 8 : Interfaçage microprocesseur/FPGA 20
Figure 9 : Flux de données entre les principaux blocs fonctionnels 21
Figure 10 : Schéma de la parallélisation de l'IDCT 25
Figure 11 : Mémoire double port 29
Figure 12 : Conception des LUT de déquantification 30
Figure 13 : Bloc arithmétique de la déquantification 30
Figure 14 : Schéma du bloc de déquantification 32
Figure 15 : Algorithme de Loeffler modifié 33
Figure 16 : Le papillon (*"butterfly"*) 33
Figure 17 : Le rotateur (*"rotator"*) 33
Figure 18 : Schéma du rotateur à 2 additions et 4 multiplications 34
Figure 19 : Organisation du processeur 35
Figure 20 : Exemple de configuration de la plate-forme 36
Figure 21 : Réglage des paramètres de la SDRAM 40
Figure 22 : Etapes du décodage MPEG-2 sur plate-forme 43
Figure 23 : Résultats du décodeur "témoin" 44
Figure 24 : Résultats du décodeur sur plate-forme 44
Figure 25 : Schéma simplifié du futur décodeur MPEG-2 complet 45

Table des tableaux

Tableau 1 : Cadences de calcul de l'IDCT et de la déquantification 24
Tableau 2 : Cadences de calcul de l'IDCT parallélisée 25
Tableau 3 : Cadences de calcul de la prédiction de mouvement brute 27
Tableau 4 : Cadences de calcul de la prédiction de mouvement parallélisée 27

1 - L'ENSEA et le laboratoire de recherche ETIS

1.1 - L'ENSEA

L'ENSEA (Ecole Nationale Supérieure de l'Electronique et de ses Applications) est une école d'ingénieurs en électronique créée en 1952 et implantée depuis 1977 à Cergy-Pontoise. Dirigée par M. Pierre Pouvil, cette école recrute à Bac+2 (prépas et DUT) ainsi qu'après un cursus universitaire (licence ou maîtrise) et dispense une formation spécialisée dans l'électronique et l'informatique.

Le corps enseignant de l'ENSEA est composé en grande partie d'enseignants-chercheurs ; ceux-ci travaillent dans les trois équipes de recherche que possède l'école, toutes trois équipes d'accueil du Ministère de l'Education Nationale :
- l'ECIME (Equipe Circuits, Instrumentation et Modélisation Electronique)
- l'ECM (Equipe Commandes et Systèmes)
- l'ETIS (Equipe Traitement des Images et du Signal).

1.2 - Le laboratoire de recherche ETIS

Le laboratoire ETIS est l'un des trois laboratoires de recherche de l'ENSEA. Ce laboratoire est commun à l'école et à l'université de Cergy-Pontoise ; c'est une équipe d'accueil du Ministère de l'Education Nationale depuis 1988 mais également une unité mixte de recherche (UMR) du CNRS depuis 2002. ETIS est membre du GdR (Groupe de Recherche) ISIS "Image / Signal" du CNRS.

Dirigée par le professeur Didier Demigny, l'ETIS s'occupe actuellement de domaines plus variés que son orientation d'origine ; pour cela, elle s'est différenciée en quatre équipes de recherche :
- l'équipe Traitement des Images
- l'équipe Traitement du Signal
- l'équipe Neurocybernétique
- l'équipe Architecture.

C'est également le laboratoire ETIS qui dispense en partie les enseignements du DEA TIS (Traitement des Images et du Signal) qui peut être suivi à l'ENSEA après une formation Bac+5, ou en parallèle avec une 3ème année à l'ENSEA.

Les effectifs de ETIS sont de :
- 5 professeurs,
- 13 maîtres de conférence et 1 ATER,
- 1 chargé de recherche CNRS,
- 1 ingénieur de recherche,
- 1 ingénieur d'études CNRS,
- 1 secrétaire,
- une vingtaine de doctorants.

1.3 - L'équipe Architecture

Cette équipe du laboratoire ETIS oriente plus particulièrement ses recherches dans le domaine de l'architecture des circuits intégrés numériques. Elle est dirigée par M. François Verdier.

Les recherches de l'équipe Architecture visent à implémenter des traitements de signaux numériques complexes (du type de ceux qu'on effectue en télévision numérique ou en télécommunications par exemple) dans des circuits intégrés dédiés de type ASIC ou FPGA, ou des circuits mixtes (processeur + FPGA) comme celui qui est utilisé dans ce projet. Elles s'organisent autour de deux axes :

Axe 1 : Architectures mixtes reconfigurables

En dehors de l'association dans un même système, voire sur un même circuit, de composants programmables et "câblés", cet axe de recherche s'intéresse au concept de "composants configurables dynamiquement" et aux architectures et outils de conception qui en découlent.

Axe 2 : Adéquation Algorithmes Architectures et IP

L'objectif de cet axe de recherche est l'étude de l'intégration sur silicium (ASIC ou FPGA) d'algorithmes non triviaux se prêtant a priori difficilement à une réalisation matérielle efficace. L'originalité de l'approche réside dans le fait que les algorithmes initiaux sont systématiquement analysés et remis en cause. L'optimisation est donc à la fois algorithmique et architecturale.

L'équipe Architecture collabore avec plusieurs laboratoires de recherche en France et à l'étranger :
LE2I : Université de Bourgogne
LPSI : ESIEE, Dépt Informatique, Marne La Vallée
LIRMM : Université de Montpellier II
LIEN : Université Henri Poincaré, Nancy
SOSSO : INRIA Rocquencourt
ENST : Laboratoire Image
Laboratoire d'électronique, Faculté des sciences de Monastir, Tunisie
Laboratoire d'informatique, Université fédérale du Minas Gerais, Brésil

2 - Mise en oeuvre du décodeur MPEG-2

2.0 - Introduction générale

Les applications multimédia actuelles (télévision numérique, etc ...) exigent des formats de compression vidéo performants. C'est la raison de la création du format **MPEG** par un comité d'experts (MPEG signifie *Moving Pictures Experts Group*, Groupe d'Experts en Images Animées, et désigne à la fois le format et ses créateurs). Ce format de compression, basé sur des principes de corrélation et d'utilisation des défauts de la vision humaine, permet une réduction significative du débit de données nécessaire à la transmission de données vidéo tout en leur conservant une qualité graphique optimale.

MPEG-2 est l'un des formats de compression définis par le comité MPEG. En raison de sa souplesse vis-à-vis des différents débits et résolutions d'images, ce format est très utilisé actuellement, notamment dans la télévision numérique et dans le stockage de données vidéo sur support dit "sans erreur" (DVD).

Les créateurs de MPEG mettent à la disposition de tous le code source d'un décodeur MPEG-2, écrit en langage C et compilable sous UNIX ou Windows. Ce décodeur peut être implémenté sur un microprocesseur, mais on subit alors les inconvénients de temps de calcul et de consommation "importante" d'énergie inhérents aux solutions purement logicielles.

Le but de l'architecture mixte est de diminuer ces inconvénients en implémentant certaines fonctions du décodeur directement sur un FPGA : on obtient ainsi un microprocesseur et un FPGA travaillant en parallèle, ce qui permet de réduire le temps de calcul de l'ensemble.

L'étude du décodeur MPEG-2 sur architecture mixte se déroule en trois étapes. Premièrement, je rappellerai les principes de la norme MPEG-2.

Dans un second temps, je présenterai le décodeur MPEG-2 logiciel du comité MPEG pour en faire l'analyse et faire ressortir les fonctions à implémenter en FPGA.

Enfin, j'indiquerai comment se déroule cette implémentation sur la plate-forme mixte, les choix retenus et les résultats qui en découlent.

2.1 - Le format MPEG-2

2.1.1 - La compression d'image JPEG

La compression vidéo MPEG est une application aux images animées de la compression d'image **JPEG** (*Joint Photographic Experts Group*) avec pertes, qu'il faut donc détailler dans un premier temps.

La compression d'image JPEG s'appuie sur un principe de redondance spatiale et de limite de visibilité des images par l'oeil humain. Elle se déroule en six étapes :

- Découpage de l'image en blocs 8x8 pixels

Chaque bloc de l'image sera traité séparément, mais certaines données sont conservées d'un bloc à l'autre, comme on le verra plus tard. Chaque bloc contient en fait trois matrices différentes : une matrice de luminance (Y), une matrice de chrominance bleue (Cb) et une matrice de chrominance rouge (Cr). La proportion de pixels de chrominance par rapport aux pixels de luminance dépend de la résolution utilisée pour l'image. En MPEG-2, la résolution principalement utilisée s'appelle **4:2:0**, c'est-à-dire 4 pixels de luminance pour 1 pixel de chrominance bleue et 1 pixel de chrominance rouge.

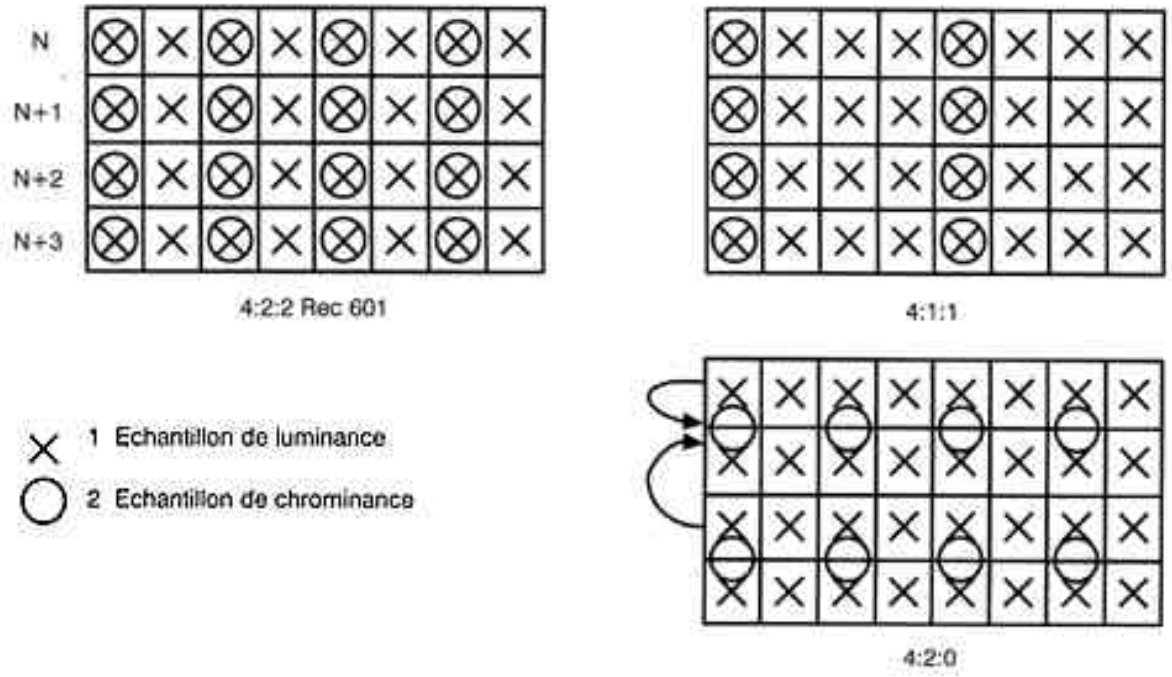


Figure 1 : Répartition des échantillons de luminance et de chrominance selon les normes

- Transformée en cosinus discrète *(Discrete Cosine Transform*, ou DCT)

Il s'agit d'une variante de la transformée de Fourier appliquée aux signaux discrets. À l'issue de cette transformation, la matrice des pixels devient une **matrice des fréquences spatiales** de l'image.

La transformée d'une séquence de longueur M est donnée par la formule :

$$C(0) = \frac{1}{\sqrt{M}} \sum s(k)$$

$$C(u) = \sqrt{\frac{2}{M}} \sum s(k) \cos[\frac{(2k+1)u\pi}{2M}]$$

où $\sum$ désigne la somme sur k allant de 0 à (M-1). La transformée de la séquence est également une séquence de longueur M.

On remarque que cette transformée est unidimensionnelle. Pour la compression JPEG (sur une matrice de pixels en 2 dimensions), on effectue 2 transformées successives, l'une sur les lignes et l'autre sur les colonnes du bloc.

Dans le cadre du décodage, on s'intéresse plutôt à la transformée inverse, appelée *Inverse Discrete Cosine Transform* ou IDCT. Elle est définie par :

$$s(k) = \frac{1}{\sqrt{M}} C(0) + \sqrt{\frac{2}{M}} \sum C(u) \cos[\frac{(2k+1)u\pi}{2M}]$$

où $\sum$ désigne la somme sur u allant de 0 à (M-1).

- Seuillage et quantification

Ce sont les deux opérations tirant profit des limites de la vision humaine, et induisant des pertes dans la compression. En effet, le seuillage réduit à 0 les coefficients inférieurs à un "seuil de visibilité" au-dessous duquel l'oeil humain est incapable de percevoir la présence de la fréquence. La quantification consiste à appliquer aux coefficients une fonction "en escalier" dont la précision diminue quand la fréquence augmente, les fréquences les plus élevées étant les moins bien distinguées par la vision humaine.

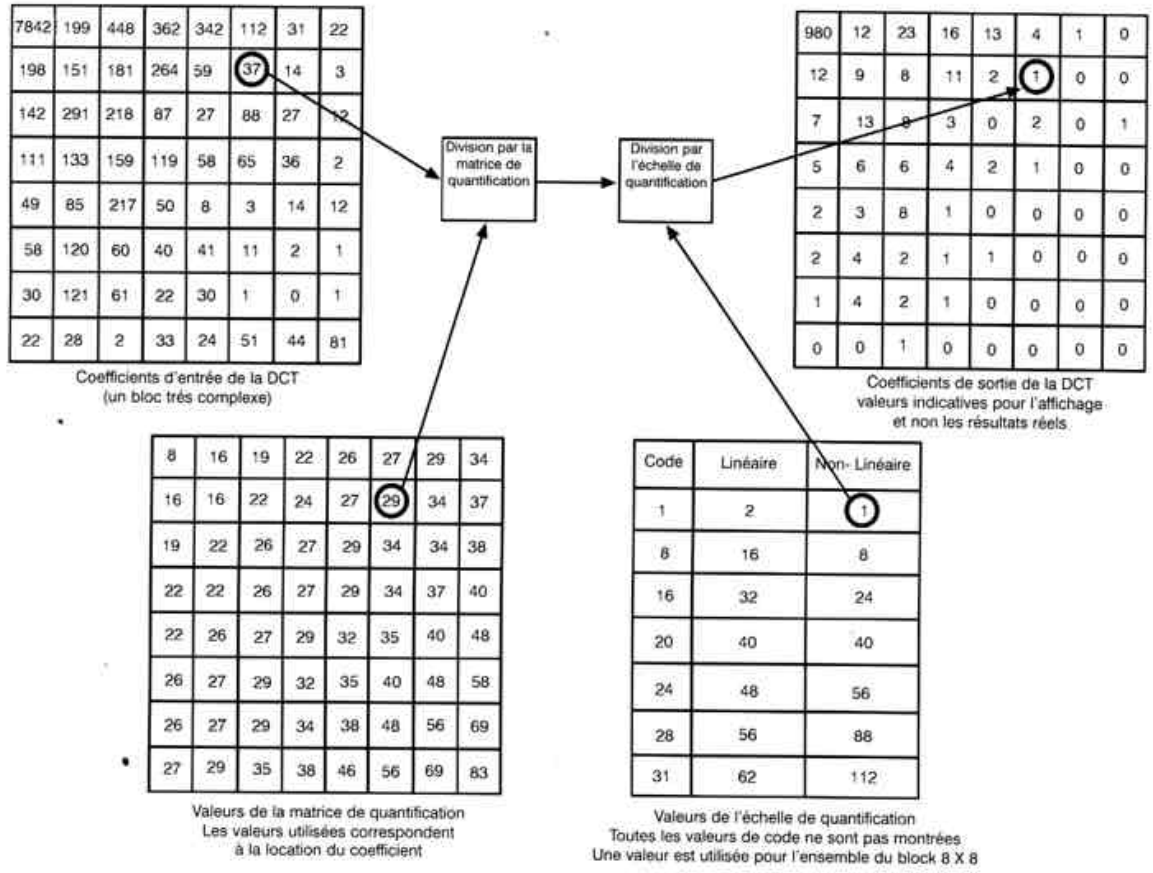


Figure 2 : Quantification d'un bloc

Lors du décodage, on opère une déquantification des coefficients. Il existe en effet des tables de déquantification, généralement contenues dans le fichier JPEG, pour réaliser cette opération. (si elles n'y sont pas, le décodeur peut utiliser une table de déquantification par défaut)

- Codage des coefficients

Le coefficient de fréquence nulle (appelé DC) est codé séparément des autres, en **DPCM** (*Differential Pulse Code Modulation*), c'est-à-dire qu'il est codé différentiellement par rapport à celui du bloc précédent pour éviter une trop grande différence de "fond" entre deux blocs successifs, cette différence étant très perceptible par l'oeil.

Les autres coefficients (appelés AC) sont lus en zigzag sur la matrice pour être convertis en une série de données. C'est ce qui permet de leur appliquer un codage RLC (*Run Length Coding*) qui tire parti du grand nombre de zéros induit par le seuillage. En effet, le RLC consiste à coder le nombre de zéros, puis la taille et la valeur du prochain coefficient non nul ; on économise ainsi d'autant plus de ressources qu'il y a de zéros dans la séquence.

Le dernier codage appliqué au bloc est un codage VLC (*Variable Length Coding*) ou codage de Huffmann. Il donne au coefficients une valeur d'autant plus courte qu'ils sont rencontrés souvent dans l'image. Comme il est difficile de prévoir pour chaque image la fréquence de rencontre d'un coefficient, il existe des tables de codage/décodage de Huffmann définies par la norme JPEG et convenant à la plupart des images.

Le décodage de l'image s'effectue également bloc par bloc en inversant les opérations (du moins celles qui sont réversibles) : il se décompose donc de la manière suivante :

- Décodage VLC (VLD, *Variable Length Decoding*)
- Déquantification
- IDCT
- Réassemblage des blocs pour former l'image

2.1.2 - La compression vidéo MPEG

La compression vidéo MPEG s'appuie, comme on a dit, sur la compression d'images fixes JPEG, mais le fait de travailler sur des images animées introduit la notion de **corrélation temporelle** qui se superpose à la corrélation spatiale déjà traitée en JPEG. La compression vidéo MPEG repose sur la **prédiction d'images** à partir des images existantes.

Parce qu'on ne peut pas prédire un mouvement à l'infini, et afin d'éviter de propager trop loin des erreurs de prédiction, les images ne sont pas prédites indéfiniment, et on revient toujours, au bout de quelques prédictions, à reprendre une image d'origine. On distingue ainsi trois types d'images différentes en MPEG :

Les images I (Intra)

Appelées ainsi car elles ne subissent aucun codage inter-images (prédiction ou interpolation) mais uniquement intra-image. Une image I est une image de la séquence vidéo originale codée directement en JPEG.

Les images P (Prédites ou *Predicted*)

Elles sont générées à partir d'une prédiction de mouvement. L'estimateur de mouvement recherche sur deux images successives un même macrobloc (ensemble carré de 4 blocs de luminance, plus les blocs de chrominance) et calcule un **vecteur de mouvement** de ce macrobloc. Un compensateur de mouvement calcule une image d'**erreur de prédiction**, et c'est l'addition de cette erreur de prédiction et du vecteur de mouvement qui constitue l'image P.

Les images B (Bidirectionnelles ou *Bidirectional*)

Ce sont les images les plus compressées car elles sont générées par **interpolation** entre les images I et/ou P qui les entourent.

Pour générer (encodeur) ou reconstituer (décodeur) ces images, il faut donc avoir en mémoire les images I et/ou P génératrices de ces images B, ce qui conduit à une modification de l'ordre des images dans le flux codé MPEG-2, qui doit être rétabli correctement au décodage.

Remarque : le format MPEG prévoit un quatrième type d'images, les images D. Elles sont peu utilisées (elles sont maintenant obsolètes) et ne seront donc pas prises en considération dans cette étude.

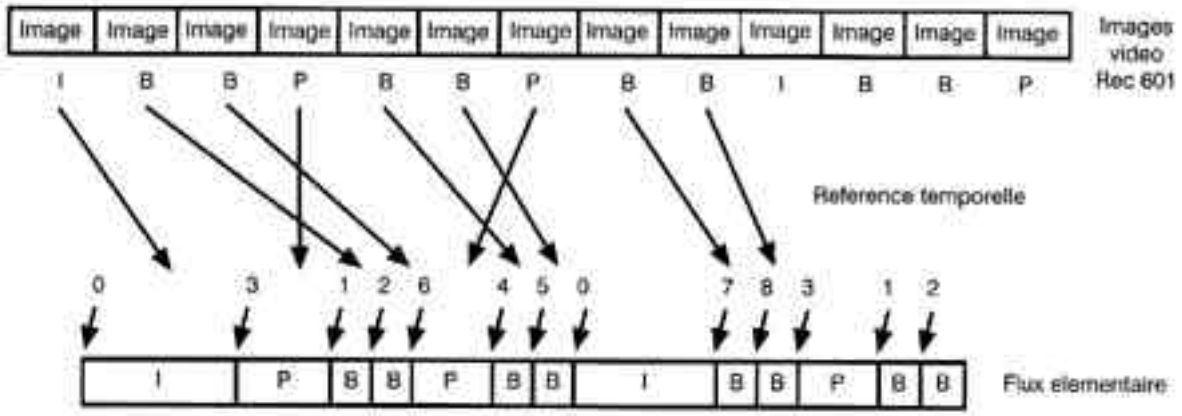


Figure 3 : Réarrangement des images dans un flux MPEG

On appelle **"groupe d'images"** une suite d'images commençant par une image I et contenant toutes les images P et B générées à partir de cette image. Une séquence MPEG est donc une succession de groupes d'images.

Le groupe d'images possède deux paramètres de taille primordiaux, notés M et N. M désigne la distance (en nombre d'images) entre 2 images P successives, tandis que N désigne la distance (en nombre d'images) entre 2 images I successives (donc la longueur du groupe d'images). La norme MPEG-1 définit généralement M=3 et N=12, soit 3 images P et 8 images B pour chaque image I. Le groupe d'images est alors :

I B B P B B P B B P B B / I ...

Décoder une séquence MPEG revient donc non seulement à décoder des images JPEG, mais également à régénérer celles-ci à partir des données de prédiction et d'interpolation. Le décodeur MPEG se base donc sur un décodeur JPEG auquel on ajoute des éléments permettant les autres opérations nécessaires au décodage.

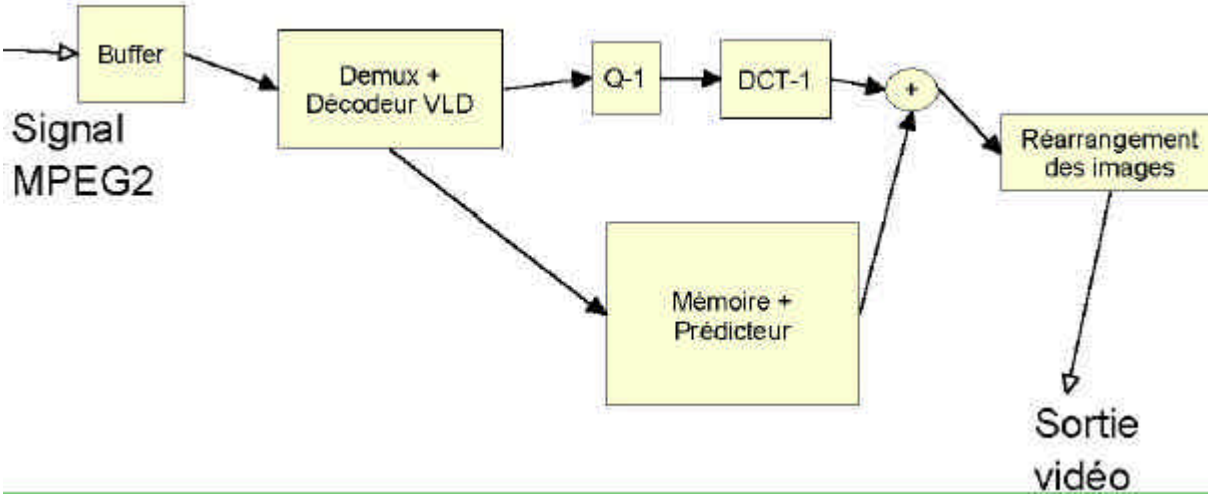


Figure 4 : Schéma simplifié d'un décodeur MPEG ou MPEG-2

Ce schéma montre les opérations nécessaires au décodage des images. Pour les images I, le décodeur JPEG simple (décodage VLD, déquantificateur et IDCT) suffit. Pour les images P et B, il y a deux éléments différents : le vecteur de mouvement est envoyé au prédicteur qui génère l'image prédite à partir de la précédente image (et de la suivante dans le cas d'une image B) ; cette image prédite est ajoutée à l'image d'erreur de prédiction traitée par le décodeur JPEG.

Ce principe de décodage est commun aux formats MPEG (MPEG-1) et MPEG-2.

On définit plusieurs niveaux d'accès aux données MPEG :

- la séquence
- le groupe d'images
- l'image
- la tranche ou *slice* (une suite de macroblocs contigus de l'image ; en MPEG-2 elle est obligatoirement sur tout ou partie d'une ligne horizontale de l'image)
- le macrobloc
- le bloc

Le décodeur étudié plus loin effectue une approche progressive du décodage en considérant chacun de ces niveaux successivement.

2.1.3 - Le format MPEG-2

Le format MPEG-2 a été conçu comme une extension de MPEG-1. Il y a **compatibilité ascendante** entre les deux, ce qui signifie qu'un décodeur MPEG-2 est capable de décoder des signaux au format MPEG-1.

MPEG-2 introduit essentiellement deux nouvelles fonctionnalités :

- Les différents niveaux et profils

MPEG-2 reconnaît cinq profils et quatre niveaux différents. Le profil correspond au jeu d'outils de compression utilisé, le niveau à la résolution des images. La norme prévoit également une **compatibilité ascendante des profils** : un décodeur est caractérisé par le profil maximum qu'il peut décoder, et fonctionne également sur tous les profils inférieurs.

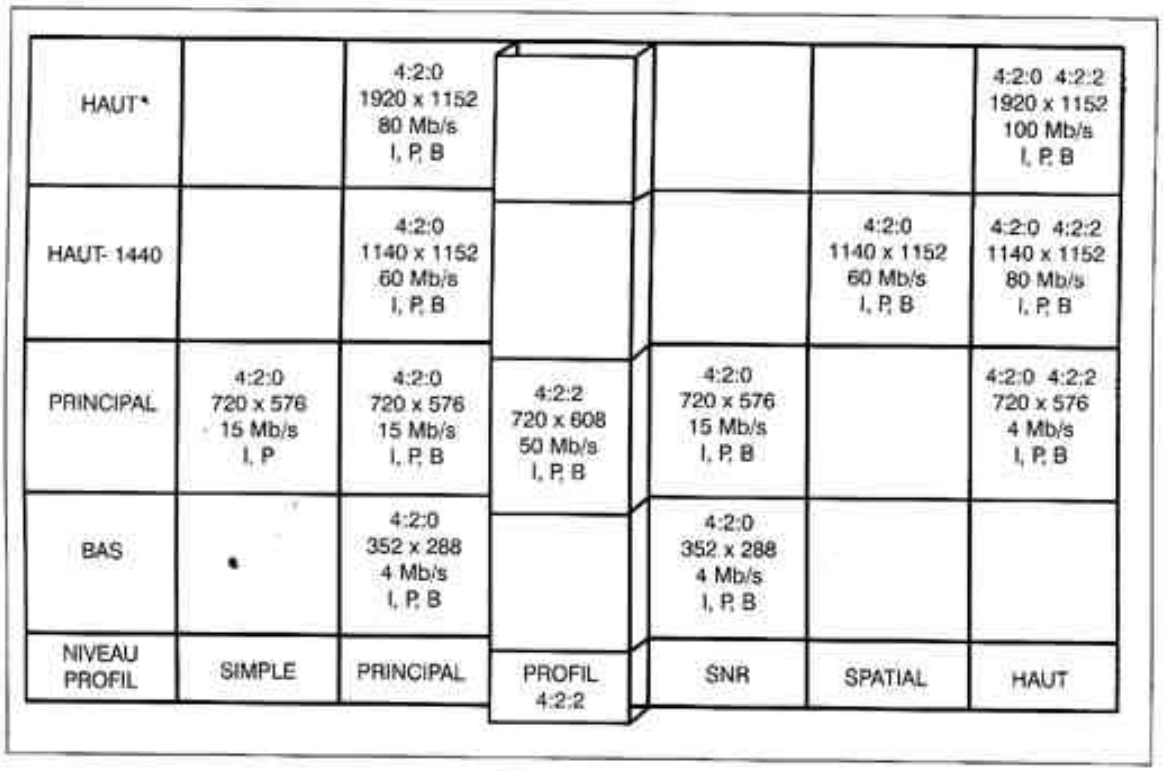


Figure 5 : Combinaisons des profils et des niveaux en MPEG-2

(dans ce schéma on a ajouté un profil supplémentaire (en relief), le profil 4:2:2)

La combinaison principale et la plus utilisée actuellement est la combinaison du niveau et du profil principaux, appelée *Main Profile at Main Level* (MP@ML), c'est-à-dire :

- *Main Level* : résolution 4:2:0 (voir plus haut) avec un format jusqu'à 720x576 pixels.
- *Main Profile* : utilisation des 3 types d'images MPEG (I, P, B) et des structures de trame.

Cette combinaison nécessite un débit de données de **15 Mbits/s** pour le flux encodé.

- Le mode trame

MPEG-2 reconnaît les images entrelacées et peut les décoder d'un bloc (mode image ou *frame mode*) ou trame par trame (mode trame ou *field mode*). Ce dernier mode s'avère très intéressant quand on traite des images en mouvement rapide, où il y a donc beaucoup de différences d'une trame à l'autre.

En mode image, les blocs et les macroblocs sont découpés dans l'image entière (formée des deux trames entrelacées). En mode trame, un macrobloc est formé de deux blocs issus de la première trame et de deux blocs issus de la seconde. Les deux trames sont alors **différenciées** pour l'étape de prédiction du mouvement.

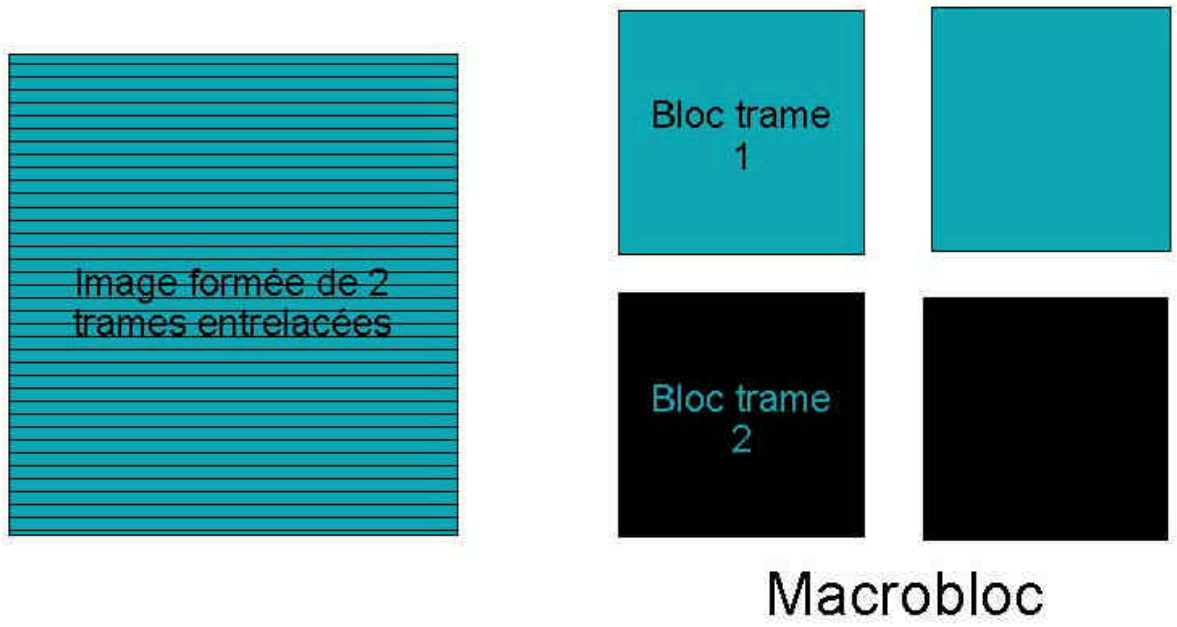


Figure 6 : Découpage des blocs en mode trame

Bien entendu, dans le cas d'une image codée en mode trame, le décodeur doit reformer les images entrelacées lors de l'étape de reconstitution de l'image.

Le schéma de principe du décodeur MPEG présenté dans la figure 4 est également valable pour un décodeur MPEG-2, mais ce dernier présente des fonctions plus complexes qui s'ajoutent aux fonctions principales ou qui les complètent, comme l'identification du profil, le décodage en mode trame ...

2.2 - Le décodeur MPEG-2 logiciel

2.2.1 - Présentation générale

La base de mon étude est un décodeur MPEG-2 purement logiciel. Ce décodeur, écrit en langage C, est disponible en téléchargement gratuit par FTP anonyme sur le serveur du MPEG à l'adresse suivante :
`ftp://ftp.mpeg.org/pub/mpeg/mssg/mpeg2vidcodec_v12.tar.gz`
(sources pour UNIX). Ces sources contiennent un encodeur et un décodeur, mais on ne s'intéresse ici qu'au décodeur.

Il s'agit de la version **1.2** du décodeur, publiée en Juillet 1996. Il n'y a pas eu d'autres versions venant du comité MPEG, car celui-ci travaille actuellement sur d'autres normes de compression vidéo destinées à la télévision numérique : le MPEG-4 et le MPEG-7.

Ce décodeur traite les profils jusqu'au type "scalable", c'est-à-dire de découpage de l'image en une couche de base et une couche d'amélioration. Il décode bien entendu (compatibilité ascendante oblige) tous les profils inférieurs, ainsi que le format MPEG-1.

Une fois compilé grâce aux fichiers Makefile fournis, le décodeur est utilisable par l'intermédiaire de la console UNIX. Il prend en entrée un fichier MPEG-2 (extension .m2v) ou MPEG-1 (extension .mpg) et fournit une sortie sous la forme précisée en ligne de commande. Cette forme peut être :
- une sortie **vidéo X11** pour l'affichage direct (non utilisée car cette sortie est en couleurs 8 bits et l'affichage des PC et des stations du laboratoire est en couleurs 24 bits) ;
- une sortie sous forme de **suite d'images** ;
- une sortie sous forme de **suite de fichiers de luminance et de chrominance**.

Exemple de ligne de commande du décodeur :

```
>./mpeg2decode -b test.m2v -o3 sortie%d
```

Le décodeur prend alors en entrée le fichier **test.m2v** et sort des images successives nommées **sortie1.ppm**, **sortie2.ppm**, etc ... (le nombre d'images de sortie est égal au nombre d'images de la séquence vidéo d'entrée)

Les principales options en ligne de commande sont :
`-b` spécifie le fichier MPEG-2 à utiliser en entrée
`-oX` spécifie le fichier de sortie et son format : o0 pour des **fichiers de luminance et de chrominance** (3 fichiers .Y, .U et .V), o1 pour des **images SIF** (.SIF), o2 pour des **images Targa TrueVision** (.tga), o3 pour des images **Portable Pix Map** (.ppm), o4 pour une **sortie X11** et o5 pour une **sortie X11 haute qualité** ; ces deux dernières options ne seront pas utilisées.
`-f` pour activer le format trame entrelacée
`-r` pour utiliser l'IDCT de référence double précision ; l'IDCT utilisée par défaut est un algorithme rapide (voir plus loin)
`-t` pour activer le mode "trace" qui affiche les données du décodage au fur et à mesure
`-vX` pour activer le mode "verbeux" (*"verbose mode"*) qui affiche le contenu des en-têtes à différents niveaux selon la valeur de X : v0 correspond à aucune donnée, v1 au niveau de la

séquence, v2 au niveau de l'image, v3 au niveau de la tranche, v4 au niveau du macrobloc et v5 au niveau du bloc.

2.2.2 - Fichiers et fonctions du décodeur

Le décodeur MPEG-2 est constitué de plusieurs fichiers C, chacun contenant plusieurs fonctions de même type :

`mpeg2dec.c` : le fichier principal contenant la fonction *main*, décodage au niveau de la séquence et du groupe d'images
`verify.c` : vérification du fichier
`gethdr.c` : vérification des en-têtes de flux
`getpic.c` : décodage au niveau de l'image, de la tranche et du macrobloc
`motion.c` : décodage des vecteurs de mouvement
`recon.c` (également appelé `predict.c`) : prédiction de mouvement
`getvlc.c` : décodage de Huffmann
`getblk.c` : décodage VLC+RLC+déquantification des blocs
`getbits.c` : fonctions de décodage au niveau des bits
`idct.c` : DCT inverse (algorithme rapide de Chen-Wang)
`idctref.c` : DCT inverse de référence double précision
`store.c` : constitution des images de sortie
`subspic.c` : fonctions utilisées en cas de substitution d'images
`spatscal.c` : fonctions spécifiques au profil *spatial scalability*
`systems.c` : routines spécifiques au système
`display.c` : fonctions d'affichage X11 (non utilisées)

Le décodeur fonctionne sur un principe de **descente progressive de niveau** : chaque fonction appelle une fonction agissant sur un niveau inférieur. Par exemple, la fonction `decode_picture` appelle la fonction `slice`, qui appelle la fonction `decode_macroblock`.

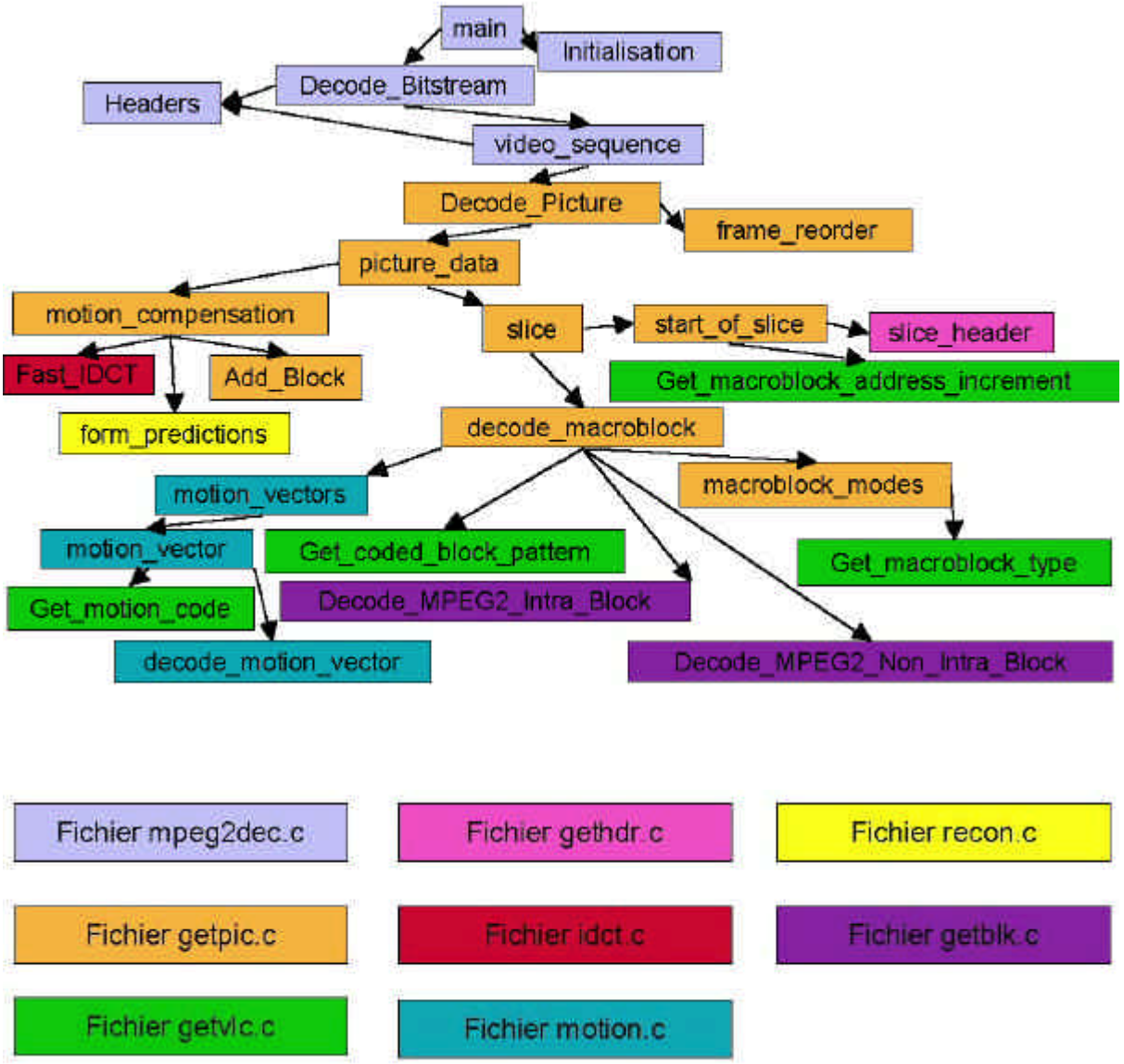


Figure 7 : Graphe d'appel des fonctions du décodeur

Remarques sur cette figure :

- Les fonctions de sortie d'images et/ou de vidéo ne sont pas représentées.
- Les fonctions spécifiques aux profils supérieurs au *Main Profile* (profils *Spatial Scalability* et *SNR Scalability*) ne sont pas représentées, on ne s'intéresse ici qu'au *Main Profile*.
- Il existe une fonction analogue à `Fast_IDCT` : la fonction `Reference_IDCT` du fichier `idctref.c` effectue une IDCT de référence double précision qui peut être utilisée dans certains cas (positionnement du *Reference_IDCT_Flag* indiqué en ligne de commande par l'option `-r`) ; cependant, la transformée utilisée par défaut est l'IDCT rapide (algorithme de Chen-Wang) et la fonction `Reference_IDCT` n'est donc pas représentée ici.
- Le bloc marqué "Initialisation" du fichier `mpeg2dec.c` contient en fait plusieurs fonctions d'initialisation des différentes fonctions du décodeur qui ne sont pas détaillées sur ce schéma.
- La fonction `Headers` appelle plusieurs fonctions du fichier `gethdr.c` qui décodent différents en-têtes et ne sont pas détaillées ici.

2.2.3 - Recherche des fonctions susceptibles d'être implémentées en hardware

On cherche pour l'implémentation en hardware des fonctions comportant des calculs simples, relativement peu nombreux et qui sont utilisées très régulièrement par le programme. Lors de l'implémentation en FPGA, on peut concevoir une fonction recevant les signaux suivants :

- l'accès à une mémoire de taille appropriée destinée à recevoir les données d'entrée et le résultat du calcul.

- un "feu vert" issu du programme C et indiquant le début du traitement ; on peut alors imaginer un FPGA agissant en séquenceur et renvoyant à son tour, si nécessaire, un signal à la fin du traitement pour indiquer au programme C que le bloc résultant est disponible en mémoire.

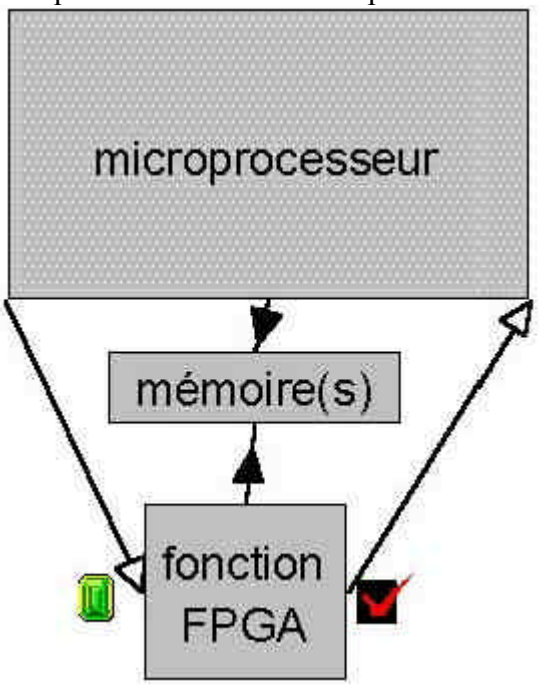


Figure 8 : Interfaçage microprocesseur/FPGA

Afin d'identifier ces fonctions, il faut voir, après le graphe d'appel des fonctions, quelles données sont passées entre les fonctions principales du programme. Pour cela, on retient les "blocs fonctionnels" suivants :

- le **décodage VLC des en-têtes** et des diverses données (fonctions du fichier `getvlc.c`)
- le **décodage VLC + RLC des blocs** (fonctions `Decode_MPEG2_Intra_Block` et `Decode_MPEG2_Non_Intra_Block` du fichier `getblk.c`)
- la **déquantification** (incluse dans les fonctions précédentes)
- la **prédiction de mouvement** (fichier `recon.c`)
- l'**addition** du bloc et de la prédiction de mouvement

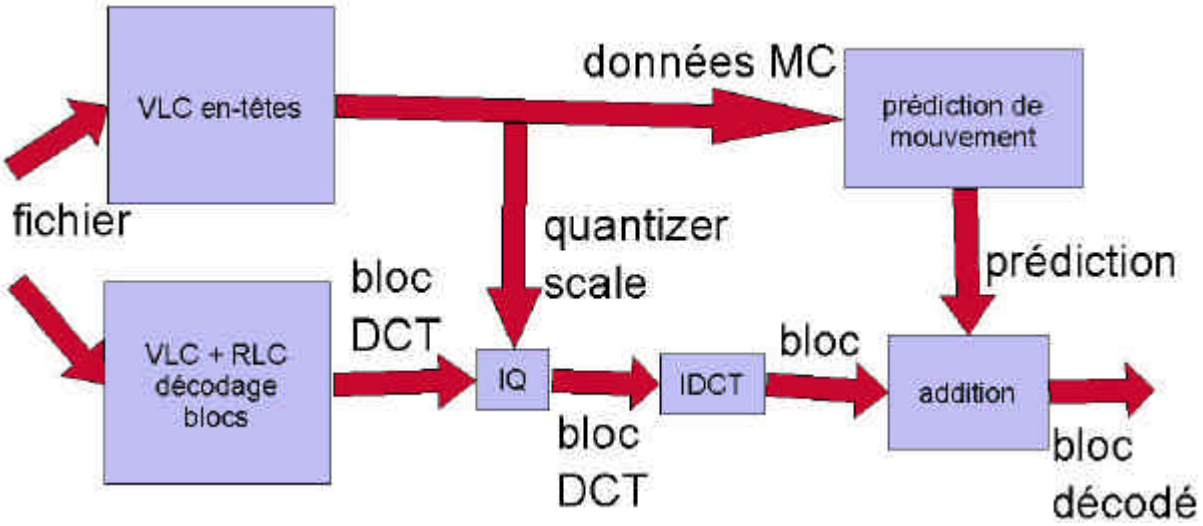


Figure 9 : Flux de données entre les principaux blocs fonctionnels

Au vu du graphe des flux de données et de son code, on peut retenir deux blocs fonctionnels en particulier : **la déquantification** et **l'IDCT**. Ces deux fonctions, en effet, effectuent toutes deux un calcul très fréquent puisqu'il est utilisé pour chaque bloc ; de plus, ce calcul est relativement simple et très régulier (il n'y a pas un nombre exorbitant de paramètres variables pour la déquantification, et pour l'IDCT, il n'y en a pas du tout). C'est donc à ces fonctions qu'il faut s'intéresser, au moins dans un premier temps, pour l'implémentation en hardware.

2.3 - Analyse des fonctions à implémenter en hardware

2.3.1 - Analyse de la fonction d'IDCT

Pour l'analyse de cette fonction comme de la suivante, on se placera dans un cas simple : celui d'un *Main Profile at Low Level* (MP@LL), soit une "basse" résolution. Ses caractéristiques sont les suivantes :

- une image de **352x288** pixels, soit **2376 blocs** (1584 blocs de luminance et 792 blocs de chrominance)
- un débit de **30 images/s**.

L'IDCT rapide est décomposée en deux fonctions analogues : l'IDCT monodimensionnelle sur les lignes du bloc (`idctrow`) et l'IDCT monodimensionnelle sur les colonnes (`idctcol`).

Flot de données en entrée et en sortie de l'IDCT

La taille de l'entrée et de la sortie est la même : un bloc de 64 *shorts*. Au format mentionné ci-dessus, l'IDCT reçoit environ **152 K*shorts*/image**, soit environ **4,5 M*shorts*/s**.

Nombre de calculs effectués par l'IDCT

Multiplications : On en compte au total **11** pour chaque ligne et chaque colonne, ce qui fait **176 multiplications/bloc** au total, soit environ **12,5 Mmultiplications/s**.

Additions : On en compte au total **34** pour chaque ligne et chaque colonne, plus **8** par colonne, ce qui fait **608 additions/bloc** au total, soit environ **43,3 Madditions/s**.

Décalages : On en compte au total **10** pour chaque ligne et chaque colonne, plus **8** par colonne, ce qui fait **224 décalages/bloc** au total, soit environ **16 Mdécalages/s**.

Accès mémoires effectués par l'IDCT

L'IDCT utilise, pour la fonction `idctcol`, une *clipping table* qui est initialisée à une valeur fixe au début du programme et qui n'est jamais modifiée par la suite : on peut donc faire de cette *clipping table* une mémoire en hardware utilisée exclusivement par l'IDCT.

On accède à cette *clipping table* 8 fois par colonne en lecture seule, soit **64 lectures/bloc**, ce qui fait environ **4,5 Mlectures/s**.

2.3.2 - Analyse de la fonction de déquantification

La déquantification est un problème plus complexe que l'IDCT. En effet, dans le programme C original, elle n'est pas effectuée par une fonction à part mais se situe à la fin de chacune des deux fonctions de décodage de blocs, `Decode_MPEG2_Intra_Block` et `Decode_MPEG2_Non_Intra_Block`. Le traitement de déquantification n'est pas exactement le même dans les deux cas, mais il est similaire.

On suppose ici qu'on extrait du programme C la déquantification dans sa plus simple expression, c'est-à-dire qu'on laisse le programme C s'occuper du *scanning* (Zig-Zag ou alternatif) du flot de données et remplir le bloc DCT quantifié avec les coefficients ; il devra également remplir un second tableau contenant les signes des coefficients, ceux-ci étant pris en compte après la déquantification.

Flot de données en entrée et en sortie de la déquantification

La sortie de la déquantification est l'entrée de l'IDCT : un bloc de 64 *shorts*. Conformément aux calculs indiqués plus haut, la déquantification sort environ **4,5 M*shorts*/s**.

L'entrée est plus complexe car il y a plusieurs données différentes.

Le bloc de valeurs est initialement codé en *ints* selon le programme C. Cependant, en pratique, les coefficients quantifiés se situent presque tous entre -7 et 7 et ne dépassent presque jamais la valeur absolue de 10. Il est donc possible de coder ce bloc simplement en octets.

Le signe est égal à 0 ou 1, il peut donc être codé simplement en bits.

Les autres paramètres sont l'échelle de quantification (*quantizer_scale*) codée sur un *int*, le type de bloc (Intra ou non) codé sur un bit (égal à 0 ou 1).

La déquantification reçoit alors **64 octets, 1 int et 65 bits par bloc**, soit au format considéré, **4,5 Moctets, 71280 ints et 154440 bits par seconde**.

Nombre de calculs effectués par la déquantification

Additions : Il y a **1 addition/valeur** *au maximum*, car cette addition n'a lieu que pour la déquantification d'un bloc non-intra. Ce qui fait **64 additions/bloc** *au maximum*, soit environ **4,5 Madditions/s** *au maximum*.

Multiplications : Il y a **2 multiplications/valeur** *au maximum*. Ce qui fait **128 multiplications/bloc** *au maximum*, soit environ **9,1 Mmultiplications/s** *au maximum*.

Décalages : Il y a **2 décalages/valeur** *au maximum*, car la déquantification des blocs intra n'en effectue qu'un seul. Ce qui fait **128 décalages/bloc** *au maximum*, soit environ **9,1 Mdécalages/s** *au maximum*.

Nombre d'accès mémoire effectués par la déquantification

On a en mémoire 4 tables de déquantification : luminance intra, luminance non-intra, chrominance intra et chrominance non-intra. Ces 4 tables sont propres à la séquence vidéo et sont contenues dans son en-tête : on considère donc qu'elles sont chargées dans une mémoire hardware lors du décodage de cet en-tête, avant tout décodage de bloc, et qu'elles n'interviennent donc pas dans le débit en entrée de la déquantification.

Pour chaque valeur, on effectue **1 lecture** de l'une de ces 4 tables. Ce qui donne **64 lectures/bloc** *au maximum*, soit **4,5 Mlectures/s** *au maximum*.

Attention ! Par "valeur", on sous-entend "valeur *non nulle*". Ce qui signifie que les maxima donnés pour le nombre de calculs et d'accès mémoire ne sont, dans la pratique, **quasiment jamais atteints**, les blocs donnés en entrée contenant, par principe, un nombre assez important de valeurs nulles ; par ailleurs, le traitement des blocs Intra comporte moins d'opérations que le traitement des blocs non-Intra, ce qui éloigne encore la réalité de cette asymptote car tous les blocs ne sont pas des non-Intras (il peut même y avoir des macroblocs Intra dans les images P)

2.3.3 - Performances "brutes" de l'IDCT et de la déquantification

Ce tableau présente le nombre de calculs opérés par l'IDCT et par la déquantification "telles quelles" en MP@LL et en MP@ML (les deux résolutions ont un débit de 30 images/s).

	MP@LL	**MP@ML**
Nombre de blocs par image	*2 376*	*9 720*
Flot de données en entrée/sortie de l'IDCT	4,5 M*shorts*/s	18,7 M*shorts*/s
Nombre d'additions de l'IDCT	43,3 Madditions/s	**177,3 Madditions/s**
Nombre de multiplications de l'IDCT	12,5 Mmultiplications/s	51,3 Mmultiplications/s
Nombre de décalages de l'IDCT	16 Mdécalages/s	65,3 Mdécalages/s
Nombre d'accès mémoire de l'IDCT	4,5 Mlectures/s	18,6 Mlectures/s
Flot de données en entrée de la déquantification	4,5 Moctets, 71280 ints et 154440 bits par seconde	18,6 Moctets, 291600 ints et 18,9 Mbits par seconde
Nombre d'additions de la déquantification	4,5 Madditions/s *au maximum*	18,6 Madditions/s *au maximum*
Nombre de multiplications de la déquantification	9,1 Mmultiplications/s *au maximum*	37,3 Mmultiplications/s *au maximum*
Nombre de décalages de la déquantification	9,1 Mdécalages/s *au maximum*	37,3 Mdécalages/s *au maximum*
Nombre d'accès mémoire de la déquantification	4,5 Mlectures/s *au maximum*	18,6 Mlectures/s *au maximum*

Tableau 1 : Cadences de calcul de l'IDCT et de la déquantification

2.3.4 - Parallélisation de l'IDCT

On voit tout de suite que dès que le profil utilisé est supérieur au MP@LL, l'IDCT, telle qu'elle est dans le programme C, demande des cadences de calcul difficiles à suivre pour un FPGA. L'implémentation de cette fonction en hardware doit donc passer par une **parallélisation des calculs** afin de diminuer la cadence imposée au FPGA.

Pour effectuer cette parallélisation, on part de la décomposition de l'IDCT 2D en 2 IDCT 1D, décomposition qui est déjà faite dans la fonction C qui divise `Fast_IDCT` en deux fonctions `idctcol` et `idctrow`. Le principe de la parallélisation consiste à découper le bloc en 8 lignes et à effectuer l'IDCT 1D sur chacune de ces lignes en parallèle. Ensuite, on reforme le bloc, on le redécoupe en 8 colonnes et on effectue de même l'IDCT 1D sur chaque colonne en parallèle. On reforme alors à nouveau le bloc qui est prêt à être sorti.

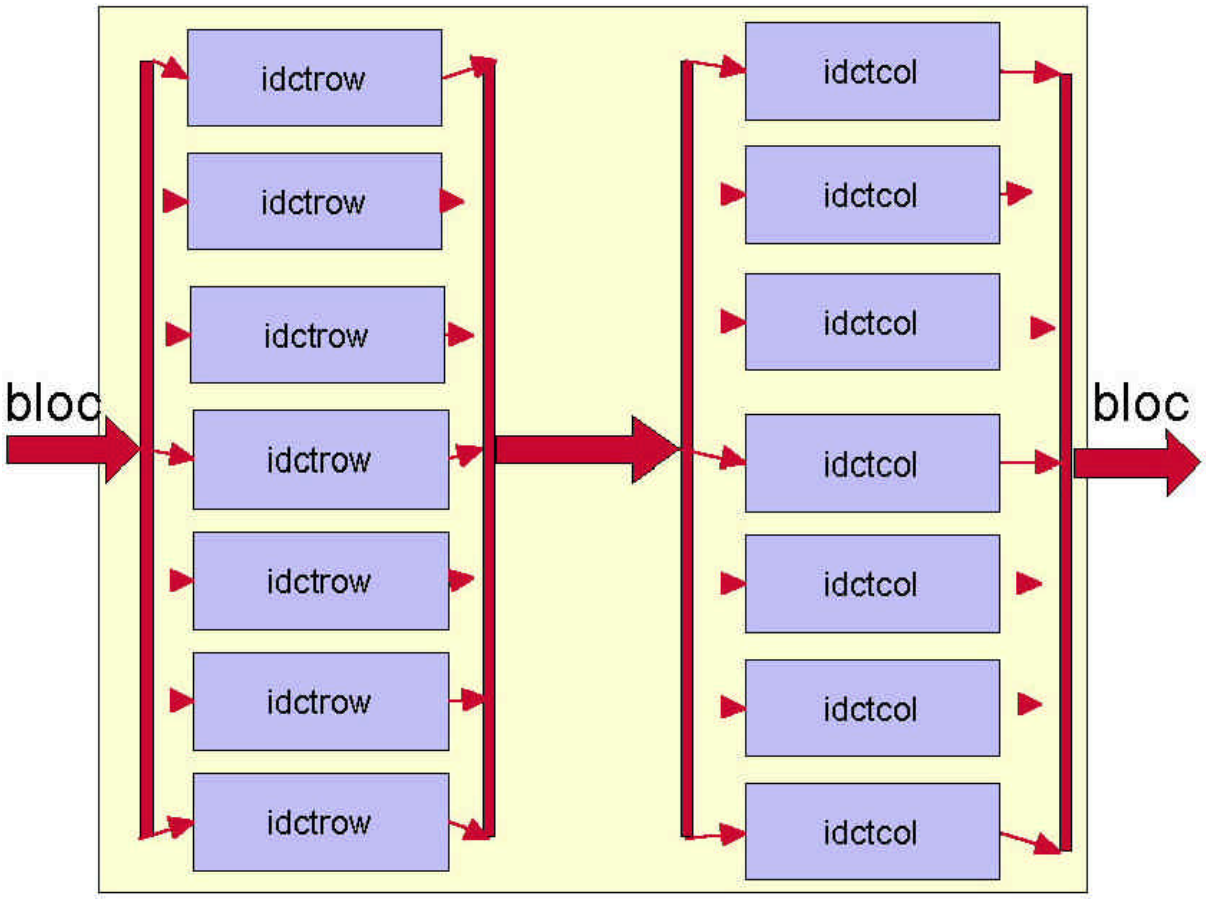


Figure 10 : Schéma de l'IDCT parallélisée

Dans ces conditions, les calculs sur les 8 lignes et les 8 colonnes se font en parallèle. Pour le rythme de calcul, on ne compte plus qu'une ligne et une colonne par bloc.

On peut alors recalculer les performances à atteindre avec une telle structure :

	MP@LL	**MP@ML**
Nombre d'additions	2,9 Madditions/s	12,2 Madditions/s
Nombre de multiplications	784 Kmultiplications/s	3,2 Mmultiplications/s
Nombre de décalages	1,3 Mdécalages/s	4,6 Mdécalages/s

Tableau 2 : Cadences de calcul de l'IDCT parallélisée

Pour ce qui est des appels mémoire, on peut discuter l'utilité de la *clipping table*. Elle a été conçue par les concepteurs du décodeur pour réaliser simplement une saturation ; cependant, dans le FPGA, on peut réaliser une **fonction de saturation** en VHDL qui remplit le même rôle qu'une lecture dans la *clipping table* sans nécessiter l'utilisation d'une mémoire.

On précise, de plus, la taille des données traitées par l'IDCT. Au vu des calculs et de la sortie de la déquantification, la précision des coefficients à l'entrée de l'IDCT s'étend sur **16 bits**. De plus, à la sortie de la saturation mentionnée ci-dessus, les coefficients sont en complément à 2 sur **9 bits**.

2.3.5 - La fonction de prédiction de mouvement

Parallèlement au travail effectué sur la déquantification et l'IDCT, on s'intéresse également aux fonctions de **prédiction de mouvement** contenues dans le fichier `recon.c`. Dans le programme C, les fonctions sont appelées l'une après l'autre, mais il est possible de paralléliser les deux car la prédiction de mouvement et l'ensemble déquantification+IDCT n'utilisent pas les mêmes données.

Alors que la déquantification et l'IDCT se basent sur une structure *layer data* (appelée `ld` dans le programme) complexe et permettant l'extraction de données du fichier d'entrée, la prédiction de mouvement se base sur des *picture frames* qui sont des tableaux d'octets de la taille d'une image de la séquence, et comprenant trois "couches" (une pour la luminance et une pour chaque chrominance).

Ces *frames* :
- sont **initialisés** au début du décodage par la fonction `Initialize_Sequence` du fichier `mpeg2dec.c` en fonction de la taille des images de la séquence ;
- sont **mis à jour** par la fonction `Update_Picture_Buffers` au début de la fonction `Decode_Picture`, donc au début du décodage de chaque image, selon le type de l'image traitée ;
- sont **remplis** par `form_predictions` (macrobloc par macrobloc) puis par `Add_Block` (bloc par bloc).

La prédiction de mouvement se fait progressivement : `form_predictions` fixe la **référence** et le **mode de prédiction** avant d'appeler `form_prediction`, qui fixe quelques paramètres en fonction du **format de couleurs** et appelle `form_component_prediction` pour chaque composante de l'image ; cette dernière s'occupe des calculs **au niveau le plus bas** (additions et décalages).

La fonction `form_predictions` calcule la position du macrobloc à traiter dans le *frame* (grâce à la donnée `MBA`, *MacroBlock Address*, passée par `motion_compensation`) et remplit un *frame* "destination" en fonction des données d'un *frame* "source". Le *frame* "destination" est toujours `current_frame` (le *frame* correspondant à l'image actuellement décodée). Le *frame* "source" peut être `forward_reference_frame` (l'image précédente, toujours le cas si l'image est de type P) ou `backward_reference_frame` (l'image suivante).

À noter que de par l'ordonnancement des images dans un flux codé MPEG-2 (voir §2.1.2 sur le sujet), le `backward_reference_frame` est toujours disponible avant l'image B à prédire.

2.3.6 - Analyse de `form_component_prediction`

La fonction `form_component_prediction` est la fonction de prédiction de mouvement qui fait les calculs de bas niveau alors que les fonctions qui l'appellent se contentent de fixer ses paramètres en fonction des données de prédiction. C'est donc cette seule fonction qui est susceptible d'être implémentée en hardware.

Flot de données en entrée de la fonction

`form_component_prediction` reçoit :
- 2 *frames* (source et destination), tableaux d'octets de taille égale à celle des images et comportant 3 couches (luminance et chrominances) ;

- 2 *ints* pour les sauts de lignes, indiquant la longueur d'une ligne dans l'image. Vu la taille de ces images dans les cas envisagés, les *ints* sont inutiles, on peut réduire le format à des *shorts* ;
- 2 *ints* pour la taille du macrobloc où se fait la prédiction. Vu ce qu'en fait la fonction `form_predictions`, c'est toujours "16 et 16" ou (plus souvent) "16 et 8", on pourrait réduire ces 2 *ints* à un seul bit ;
- 2 *ints* pour les coordonnées du 1er échantillon du macrobloc. Ces coordonnées sont calculées à partir de `MBA` et elles sont forcément inférieures aux dimensions des images traitées : aux formats étudiés ici, on peut donc se contenter de *shorts* ;
- 2 *ints* pour le vecteur de mouvement ; dans la pratique ces valeurs sont des nombres signés dont la valeur absolue maximale est aux alentours de 20, on peut donc réduire le format à des octets ;
- 1 bit pour l'*average_flag*.

Nombre de calculs effectués par la fonction

Le nombre de calculs dépend de la nature du vecteur de mouvement (composantes horizontale et verticale) et de la présence ou non de l'*average_flag* ; si celui-ci est actif, le *frame* de destination est additionné à la prédiction, sinon, il est remplacé par celle-ci.

On se place dans le cas contenant le plus de calculs, c'est-à-dire **6 additions** et **2 décalages** par valeur du macrobloc. Sachant qu'un macrobloc inclut les données de luminance et de chrominance, cela donne **2304 additions** et **768 décalages** par macrobloc. Dans tous les cas, il n'y a pas de multiplications dans la fonction.

	MP@LL	**MP@ML**
Nombre de macroblocs par image	*396*	*1 620*
Nombre max d'additions	27,4 Madditions/s	**111,9 Madditions/s**
Nombre max de décalages	9,1 Mdécalages/s	37,3 Mdécalages/s

Tableau 3 : Cadences de calcul de la prédiction de mouvement brute

2.3.7 - Parallélisation de la prédiction de mouvement

La prédiction de mouvement, comme l'IDCT, peut donner lieu à des cadences de calcul élevées en MP@ML. Cependant, elle est facile à paralléliser.

Le premier niveau de parallélisation consiste à traiter les trois composantes du *frame* (qui sont indépendantes) en parallèle. La cadence maximum à atteindre est donc celle du calcul pour la luminance, les deux chrominances étant traitées en parallèle.

On peut envisager un second niveau de parallélisation beaucoup plus poussé : le calcul de chaque valeur du *frame* destination dépend uniquement de cette valeur. Il est donc possible de calculer toutes les valeurs en parallèle. Le nombre de valeurs dans un macrobloc est toujours le même (il ne dépend pas de la taille de l'image).

Avec le premier niveau de parallélisation, on obtient les résultats suivants :

	MP@LL	**MP@ML**
Nombre max d'additions	18,2 Madditions/s	74,6 Madditions/s
Nombre max de décalages	6,1 Mdécalages/s	24,9 Mdécalages/s

Tableau 4 : Cadences de calcul de la prédiction de mouvement parallélisée *(1er niveau de parallélisation)*

Si on implémente la prédiction de mouvement en hardware, il faut prévoir la gestion des données nécessaires à cette fonction, autrement dit des *picture frames*. Pour chaque image à décoder il y a soit deux, soit trois *frames* (sachant qu'il n'y a pas de prédiction de mouvement sur les images I) : le `current_frame` (images P et B), le `forward_reference_frame` (images P et B) et le `backward_reference_frame` (images B), qui sont chacun de la taille d'une image.

La gestion de ces *frames* peut se faire de la manière suivante :

- au **début du décodage** de chaque image, la fonction `Update_Picture_Buffers` remet les *frames* à jour puis ceux-ci sont stockés dans une mémoire hardware ;
- la fonction `form_component_prediction` implémentée en hardware **remplit** le `current_frame` en fonction des deux autres *frames* (ou d'un seul si l'image est une image P) ;
- la fonction `Add_Block` peut être également implémentée en hardware et **ajoute** la prédiction de mouvement avec les données issues de l'IDCT dans le `current_frame` ;
- les *frames* complétés sont **renvoyés au microprocesseur** qui s'occupe de la réorganisation et de la sortie des images.

Au vu de la capacité des mémoires qui font la liaison entre le processeur et le FPGA, ce type de gestion sera impossible. Il faudra envisager un transfert **macrobloc par macrobloc**.

2.4 - Conception des fonctions hardware

2.4.1 - Conception de l'ensemble IQ+IDCT

On s'intéresse en premier lieu aux fonctions de déquantification et d'IDCT car ces deux fonctions agissent au niveau du bloc et ressemblent beaucoup à leurs modèles logiciels.

L'ensemble IQ+IDCT repose sur :
- l'utilisation de **mémoires à double port** : ces mémoires sont analogues à des RAM classiques mais elles ont deux ports de lecture/écriture, ce qui permet par exemple qu'un bloc fonctionnel écrive dedans et qu'un autre bloc fonctionnel lise les données. La seule opération impossible sur une telle mémoire est une double écriture à la même adresse ;

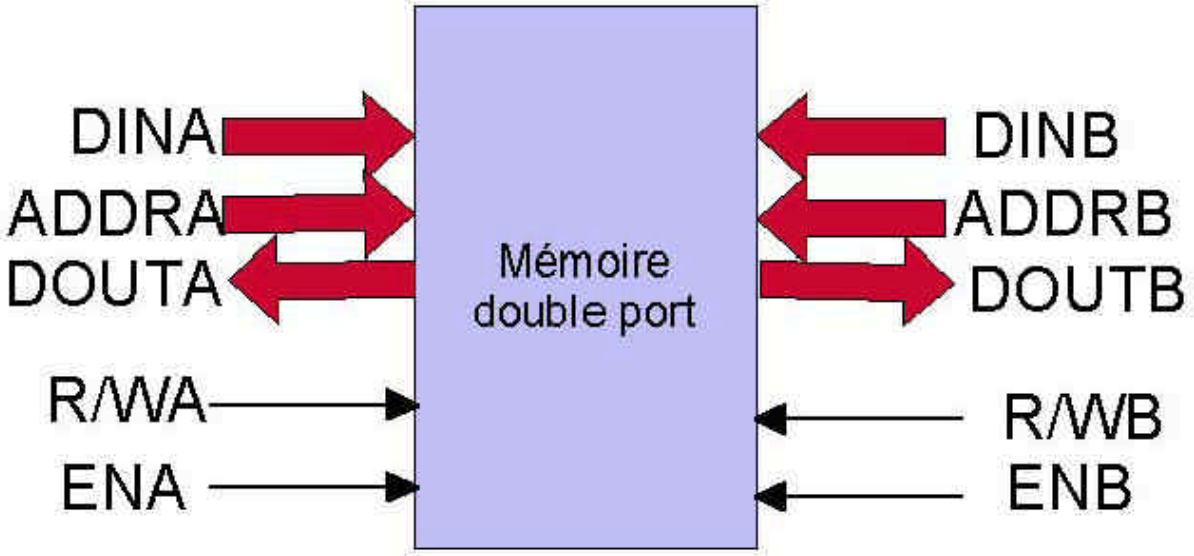


Figure 11 : Mémoire double port

- l'utilisation d'un **séquenceur** (machine d'états) destiné à coordonner les opérations, en particulier pour la synchronisation entre le FPGA et le microprocesseur.

2.4.2 - Conception de la déquantification

L'opération de déquantification proprement dite peut se décomposer en deux blocs :
- une **LUT (*Look-Up Table*)** composée des tables de déquantification ;
- un **bloc arithmétique** effectuant les opérations de déquantification en utilisant le coefficient quantifié, son signe, le produit de déquantification issu de la LUT et le *quantizer_scale*.

La **LUT** est constituée de **deux tables de déquantification** de 64 octets chacune, soit **128 octets** en tout. Deux tables seulement sont nécessaires car il apparaît, en examinant la fonction de chargement des tables de déquantification contenue dans le fichier `gethdr.c`, que les tables de déquantification des blocs de chrominance sont les copies conformes des tables de déquantification des blocs de luminance : le produit de déquantification obtenu sera donc le même qu'on utilise les tables de luminance ou de chrominance. On n'a alors plus besoin de savoir si le bloc traité est un bloc de luminance ou de chrominance.

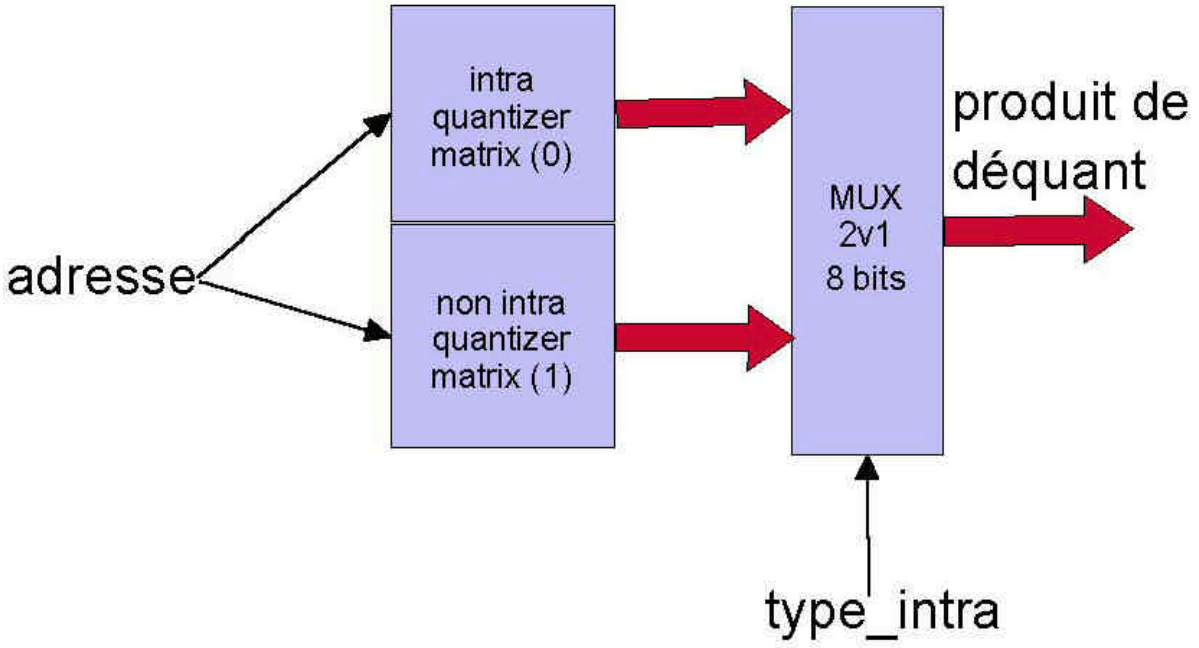


Figure 12 : Conception des LUT de déquantification

Sur cette figure, *type_intra* est une donnée d'1 bit passée par le microprocesseur pour identifier le bloc traité comme Intra (*type_intra* = 0) ou non-Intra (*type_intra* = 1). Cette donnée est d'ailleurs utile à plusieurs des fonctions de l'opération de déquantification, notamment pour le bloc arithmétique décrit ci-dessous. Le chargement des matrices de déquantification n'est pas indiqué sur la figure : lors de la lecture de ces matrices par le logiciel, le microprocesseur devra charger leurs valeurs dans les deux mémoires du FPGA.

Le **bloc arithmétique** est constitué d'un additionneur, de deux multiplicateurs, de deux décalages, d'un multiplexeur et d'un démultiplexeur ; son traitement dépend de *type_intra* et il est conçu selon le schéma suivant :

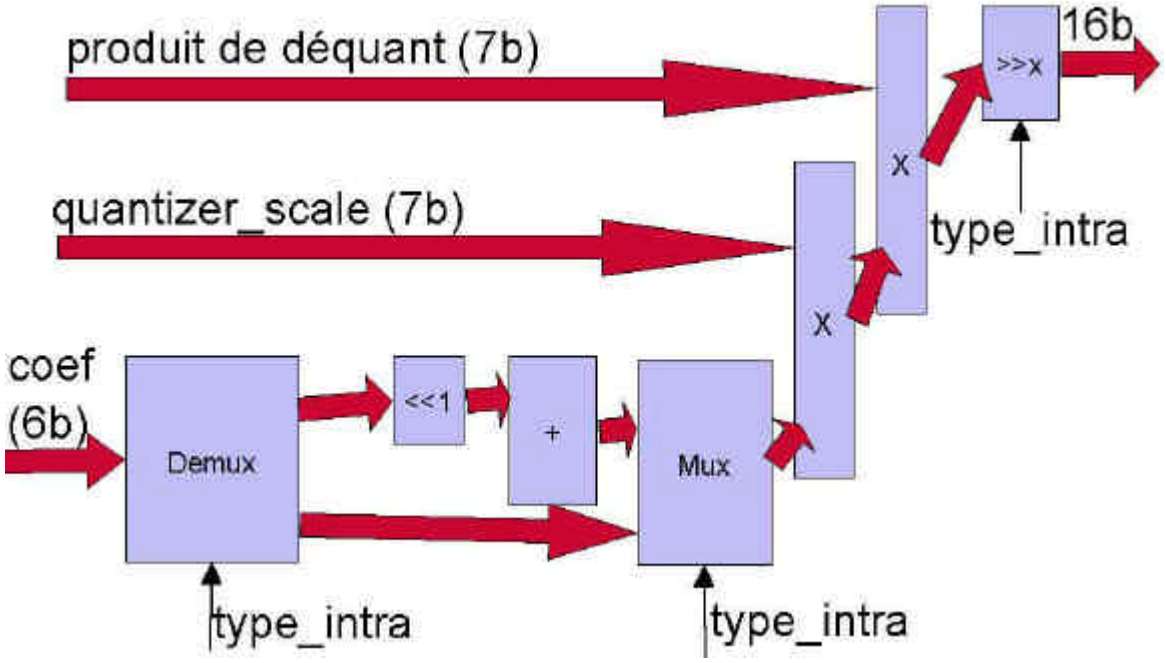


Figure 13 : Bloc arithmétique de la déquantification

Ce bloc fait les calculs en **valeur absolue**, il faut y ajouter l'incorporation du signe.
Le déquantificateur ainsi conçu traite les données séquentiellement.

La déquantification est directement connectée au microprocesseur, par l'intermédiaire d'une **mémoire à double port** (présentée dans la figure 11). Le microprocesseur doit remplir cette mémoire des valeurs à déquantifier, c'est-à-dire les valeurs non nulles du bloc décodé par la fonction `Decode_MPEG2_(...)_Block`. Ces valeurs possèdent trois composantes :

Coefficient (6 bits)	**Adresse dans le bloc (6 bits)**	**Signe du coefficient (1 bit)**

Soit un total de **13 bits**.

Même si on sait que dans la plupart des cas, le bloc contient au moins une valeur nulle (généralement le codage donne un grand nombre de valeurs nulles), la mémoire est conçue pour fonctionner dans le pire des cas, elle a donc une taille d'au moins **64 mots de 13 bits**.

La première phase de la déquantification consiste donc à lire cette mémoire et à la réinitialiser pour que le microprocesseur puisse passer au bloc suivant. Cette tâche est effectuée par un **bloc de transfert**. Ce bloc vide les données de la mémoire vers une seconde mémoire double port destinée à la déquantification.

Une fois ce transfert terminé, le FPGA peut envoyer au processeur un signal de type IRQ (interruption) pour lui signaler qu'il peut continuer le traitement, c'est-à-dire décoder un nouveau bloc ou, si le macrobloc est terminé, faire des calculs entre deux macroblocs (calcul du vecteur de mouvement, du *quantizer_scale* etc). Pendant ce temps, dans le FPGA, un **bloc de répartition séquentielle** répartit les coefficients de la seconde mémoire vers le déquantificateur au rythme des calculs de ce dernier. C'est pourquoi dans la conception de la déquantification, il faut estimer avec précision la durée des calculs du déquantificateur.

L'opération de déquantification est contrôlée par un **séquenceur**. Celui-ci peut être dans les états suivants :

- **attente**, au début du décodage ou entre deux macroblocs ;
- dès que le signal EOB arrive du microprocesseur, **activation du bloc de transfert** ;
- si on est en présence d'un bloc Intra, **chargement du coefficient DC** ;
- à la fin du transfert, **signal d'acquittement** au microprocesseur et **activation du bloc de répartition** ;
- dès que tous les coefficients ont été déquantifiés, **signal de fin de bloc** pour avertir l'IDCT ou le microprocesseur que le bloc déquantifié est prêt.

Le bloc de déquantification a donc le schéma suivant :

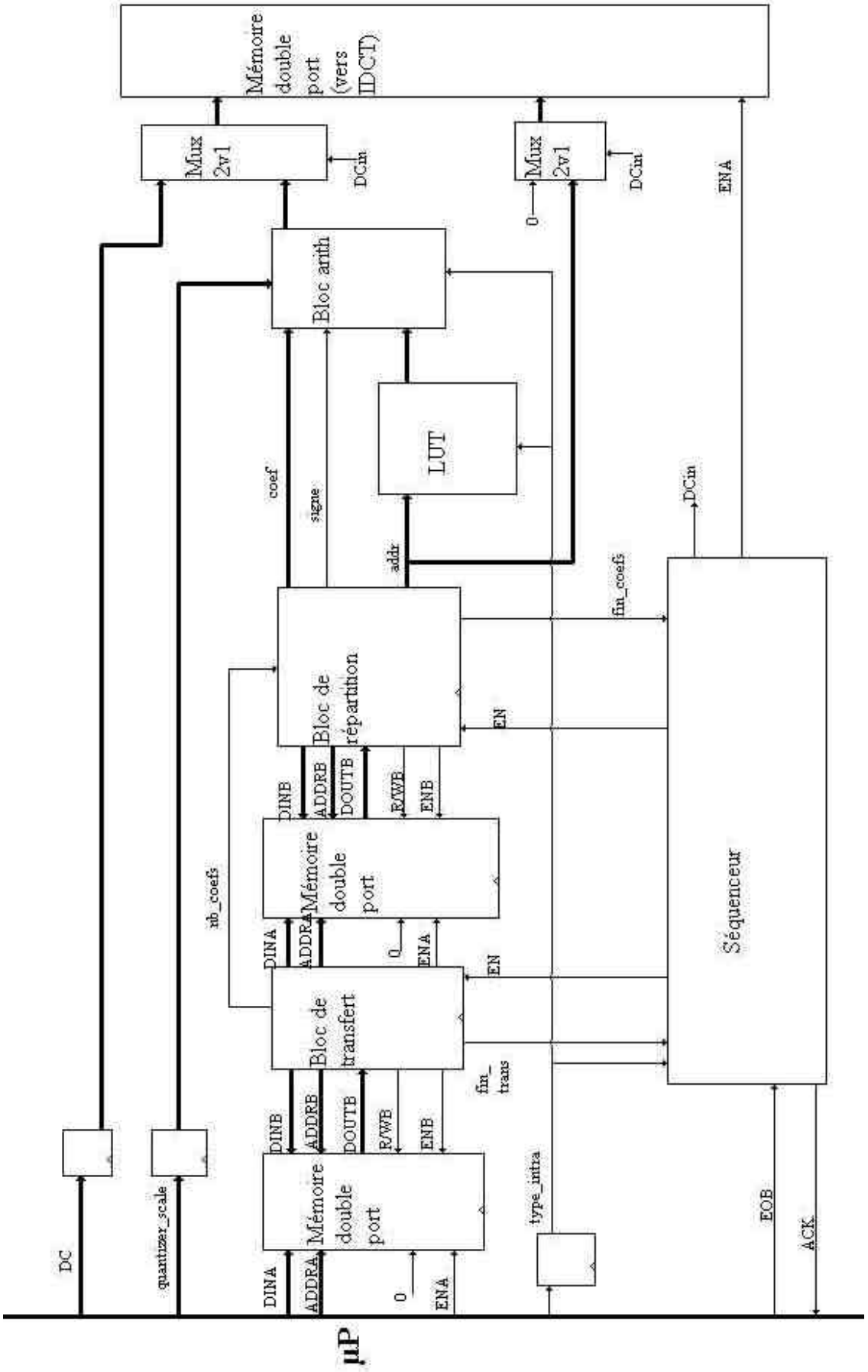


Figure 14 : Schéma du bloc de déquantification

En l'absence d'IDCT derrière la déquantification, le séquenceur envoie au microprocesseur une **interruption** qui lui demande de charger les valeurs de la mémoire de sortie dans la structure de données qu'il a prévue pour stocker les coefficients du bloc en vue de l'IDCT (la *layer data*). Si l'IDCT est implémentée, l'opération de déquantification se termine lors du stockage des coefficients et c'est l'IDCT qui prend le relais.

2.4.3 - Conception de l'IDCT

Comme on l'a vu précédemment (*paragraphe 2.3.4 : Parallélisation de l'IDCT*), on peut réduire de manière importante le nombre de calculs de la DCT inverse en la parallélisant : le bloc déquantifié subira alors 8 opérations en parallèle sur ses lignes (*idctrow*) puis 8 opérations en parallèle sur ses colonnes (*idctcol*). On peut également améliorer les performances de l'IDCT en choisissant un algorithme alliant simplicité, rapidité et précision.

Or, l'opération de DCT inverse est bien connue des concepteurs de fonctions hardware. Plusieurs implémentations de cette fonction en FPGA ont déjà été faites dans des buts divers *(voir la bibliographie pour quelques articles de recherche traitant le sujet)*. L'algorithme couramment utilisé pour ces implémentations est l'algorithme dit **"de Loeffler modifié"**. Il se représente de la manière suivante :

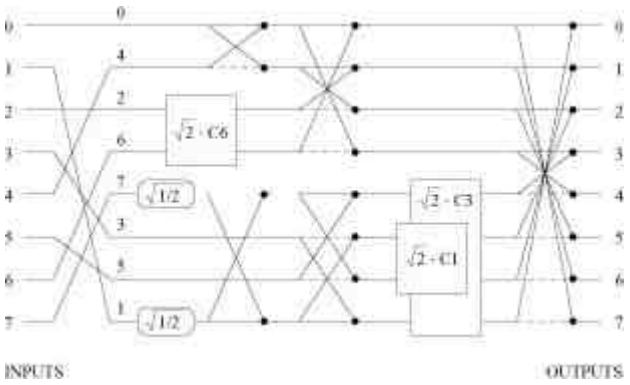
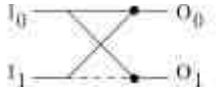


Figure 15 : Algorithme de Loeffler modifié

Comme on le voit sur la figure 15, l'algorithme de Loeffler modifié utilise trois types d'opérations différentes :

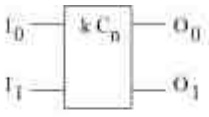
- un **multiplicateur** par une constante ;
- un **papillon (*"butterfly"*)** dont le schéma est donné ci-dessous :



$$O_0 = I_0 + I_1$$
$$O_1 = I_0 - I_1$$

Figure 16 : Le papillon (*"butterfly"*)

- un **rotateur (*"rotator"*)** dont le schéma est donné ci-dessous :



$$O_0 = I_0 k \cos\frac{n\pi}{16} - I_1 k \sin\frac{n\pi}{16} = C'_n I_0 - S'_n I_1$$
$$O_1 = I_0 k \sin\frac{n\pi}{16} + I_1 k \cos\frac{n\pi}{16} = S'_n I_0 + C'_n I_1$$

Figure 17 : Le rotateur (*"rotator"*)

Ce rotateur se compose de **2 additions et de 4 multiplications**. Il existe d'autres manières de le réaliser avec 3 multiplications et 3 additions, mais la structure à 2 additions et 4 multiplications offre un chemin critique plus court et une meilleure précision de calcul.

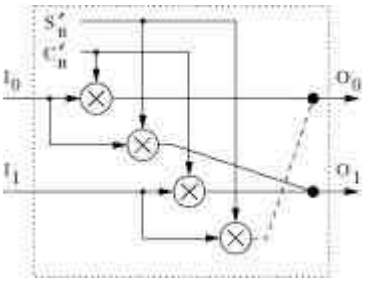


Figure 18 : Schéma du rotateur à 2 additions et 4 multiplications

L'entrée de l'IDCT comporte 16 bits en complément à 2 ; or, les coefficients décodés doivent être des coefficients de 8 bits signés. Il faut donc, à l'issue de l'IDCT, prévoir une fonction de saturation des coefficients à 8 bits.

L'IDCT, tout comme la déquantification, est pilotée par un **séquenceur**. Celui-ci permet de déclencher les différentes étapes de l'IDCT successivement :

- dès que la mémoire double port partagée avec l'IDCT est prête, le séquenceur déclenche un **bloc de transfert** qui distribue le contenu de la mémoire vers les entrées du bloc d'*idctrow* ;
- à la fin de ce transfert, il active l'idctrow et l'*idctcol* ;
- lorsque l'IDCT est terminée, il déclenche un second **bloc de transfert** qui "range" le bloc résultant de l'IDCT dans une mémoire double port connectée soit au microprocesseur, soit à la fonction `Add_Block` si celle-ci est implémentée en hardware.

3 - Résultats de l'implémentation sur la plate-forme

3.1 - La plate-forme mixte

3.1.1 - Présentation

La plate-forme mixte utilisée pour le projet est un Virtex-II Pro de Xilinx. Celle dont on dispose au laboratoire est composée d'un FPGA Virtex-II dans lequel est intégré un processeur Power-PC 32 bits.

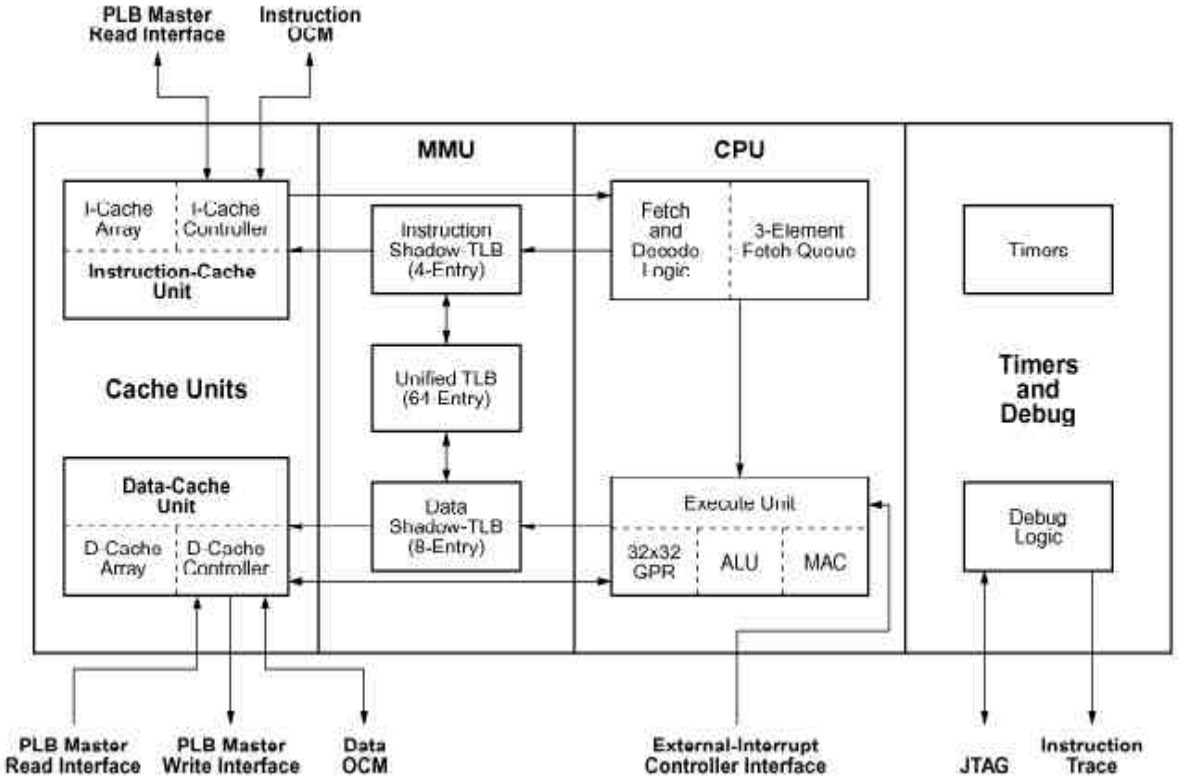


Figure 19 : Organisation du processeur Power-PC 32 bits

Ce processeur, comme on le voit sur la figure, possède deux interfaces mémoire. La première est l'OCM (*On-Chip Memory*), partagée en deux blocs distincts pour les données (*data OCM*) et les instructions (*instruction OCM*), et qui ne sera pas utilisée dans le cadre de ce projet. La seconde est l'interface PLB, sur laquelle peut être connectée une **RAM (*Block RAM*, BRAM)** d'environ 80 Ko maximum, une **SDRAM** de 32 Mo, ou les deux.

Le processeur (ainsi que le reste de la plate-forme) est programmé par une interface **JTAG**. Le matériel utilisé pour le JTAG au laboratoire est un câble qui se branche sur le port parallèle du PC hôte et sur l'interface *"Parallel Cable IV"* (câble parallèle de type IV) de la plate-forme.

Les autres unités préconçues qui entourent le processeur Power-PC sont les suivantes :

- un **bloc de reset** destiné à propager le signal de réinitialisation du système ;
- une unité série RS-232 **UARTlite** destinée à établir la communication par liaison série avec le PC hôte, par l'intermédiaire du logiciel de communication HyperTerminal ;

- quatre **LEDs** de contrôle ;
- un interrupteur **DIP** ;
- trois **boutons-poussoirs**.

Ces unités sont préconçues par le fabriquant, mais il est également possible de créer ses propres composants hardware et de les intégrer à un projet sur la plate-forme.

La plate-forme dispose de deux bus principaux :
- le bus **PLB** (*Processor Local Bus*) destiné au processeur lui-même et à sa périphérie ; il existe des BRAM sur les deux bus ;
- le bus **OPB** (*On-chip Peripheral Bus*) destiné aux blocs plus éloignés ; l'UARTlite n'est disponible que sur le bus OPB.

3.1.2 - Programmation de la plate-forme mixte

On considère pour l'instant qu'on n'ajoute pas de composants hardware personnalisés et qu'on veut seulement faire tourner un programme en C sur la plate-forme. Il faut pour cela sélectionner les éléments dont on a besoin. Un tutorial sur l'utilisation de la plate-forme est fourni par Xilinx ; on y trouve une configuration exemple incluant les éléments cités dans le chapitre précédent placés de la manière suivante :

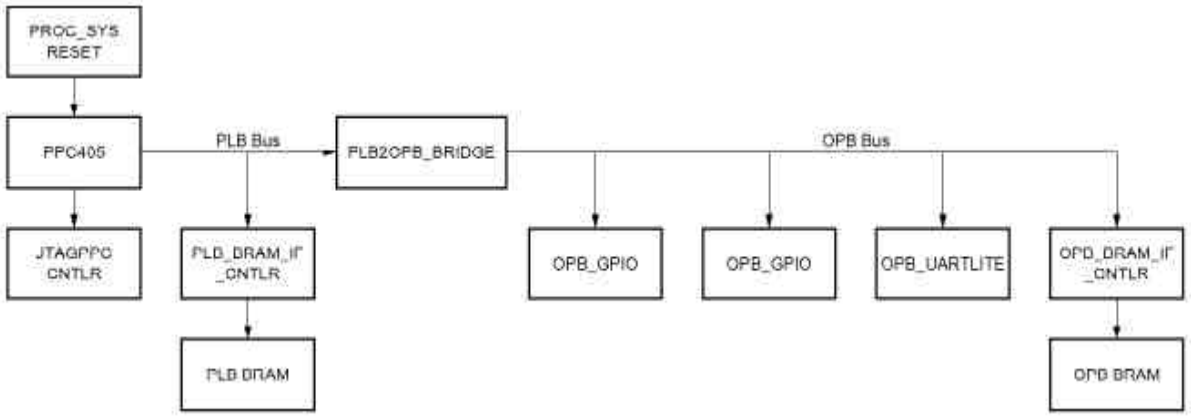


Figure 20 : Exemple de configuration de la plate-forme
(tutorial d'utilisation du Virtex-II Pro)

Les outils nécessaires à la programmation et à la configuration de la plate-forme sont fournis par Xilinx. Ils sont au nombre de 2 :

- **Xilinx Embedded Development Kit (EDK)** pour la mise en place du projet, l'ajout et la compilation des codes sources ;
- **Xilinx ISE Project Navigator** pour la synthèse et le routage des composants hardware.

Il faut ajouter à ces outils le logiciel de communication **HyperTerminal** qui permet d'établir une communication entre le PC hôte et la plate-forme par l'intermédiaire de l'UARTlite et du port série.

Les fonctions "classiques" d'entrées-sorties, `scanf` et `printf`, ne peuvent pas être utilisées sur la plate-forme en raison de leur taille importante. Les bibliothèques C de Xilinx

contiennent une fonction équivalente à `printf` mais de taille moindre, `xil_printf`. La question de l'équivalence à `scanf` sera abordée plus loin.

Des tests ont été effectués avec EDK et ISE Project Navigator sur un programme encore plus simple que celui-ci, un *"hello world"*, afin de prendre en main ces outils et de vérifier certains paramètres. La mise en place d'une architecture logicielle (sans ajout de blocs hardware personnalisés) passe par les étapes suivantes :

- choix de la plate-forme cible et des éléments à incorporer (Power-PC, contrôleur JTAG, RAM, UART etc ...) ;
- génération de la netlist ;
- exportation vers ISE Project Navigator et création du fichier de configuration par synthèse du design ;
- génération et compilation des bibliothèques logicielles ;
- compilation des sources C ;
- réunion du fichier de configuration et de l'exécutable ;
- téléchargement vers la plate-forme.

(Voir les annexes consacrés à EDK et ISE pour plus de détails.)

Le test du *"hello world"* a permis de mesurer le baudrate maximum admissible par la liaison série dans les conditions du laboratoire : il est de **115 200 bauds**.

3.2 - Gestion de la mémoire

3.2.1 - Chargement d'un programme dans la SDRAM

Le décodeur MPEG-2 étant un logiciel très dense, les parties de ce décodeur consacrées aux profils de type *scalable* ont été supprimées afin de ne traiter que le *Main Profile*. Cependant, même dans ces conditions, le programme obtenu est trop volumineux pour tenir sur la BRAM de la plate-forme, dont la taille ne peut aller que jusqu'à 80 Ko environ ; de plus, une utilisation trop importante de cette BRAM empêcherait de créer les blocs mémoire "tampons" (mémoires double port) indispensables à la connexion entre le microprocesseur et le FPGA, qui utilisent les mêmes ressources matérielles que les blocs RAM. Heureusement, il est possible de faire tourner un programme **chargé dans la SDRAM** : pour cela, il faut utiliser le *Xilinx Microprocessor Debugger (XMD)* et sa fonction `xdownload`, qui permet le téléchargement de données ou d'exécutables à une adresse valide (qui peut donc être celle de la SDRAM). Pour cela, il faut respecter certaines étapes :

- le processeur doit être **démarré au préalable** avec un programme minimal chargé dans la BRAM car il ne peut pas *booter* directement dans la SDRAM ;
- les sources du programme destiné à la SDRAM doivent être compilées hors de EDK par le compilateur destiné au Power-PC, *powerpc-eabi-gcc*, appelé par la console *xygwin*, avec un *linker script* indiquant que le programme doit être envoyé à l'adresse de la SDRAM avec toutes les sections qui lui sont nécessaires (pile, données initialisées ou non) ;
- le programme doit être téléchargé par la fonction `xdownload` du *XMD* ;
- il faut utiliser la fonction `xcontinue` du *XMD* qui ordonne au microprocesseur d'exécuter son programme à partir de l'adresse qu'on lui passe ; en l'occurence, cette adresse doit être celle de la zone de boot de l'exécutable téléchargé en SDRAM. Dans la pratique, cette adresse peut être omise car le téléchargement de l'exécutable par `xdownload` replace automatiquement le pointeur de commande (PC) sur la zone de boot de l'exécutable.

(Voir l'annexe consacré à XMD et à la console xygwin pour plus de détails.)

La SDRAM est ainsi divisée en deux parties : l'une contenant l'exécutable, l'autre abritant le système de fichiers utilisé par le décodeur.

3.2.2 - Le *linker script*

Le fichier de *linker script* est un élément important du projet. En effet, ce script, passé au *linker* lors de la compilation des fichiers sources (directement par EDK ou indépendamment sur la console *xygwin*), contient **l'organisation** des différentes sections de code dans la mémoire utilisée.

Au début du *linker script* sont déclarées les différentes zones mémoire disponibles, avec leur adresse de base et leur taille. Une zone doit être obligatoirement déclarée (en plus de la mémoire disponible) : la **zone de boot**. Cette zone fait 4 octets de longueur et se place généralement à la fin de la mémoire. Elle formera la première instruction qui devra être lue par le Power-PC lors de l'exécution du programme. À noter que si la taille de la mémoire déclarée est inférieure à la taille de l'exécutable, la compilation se termine par une erreur de type *"<memoire> is full"* et l'exécutable n'est pas généré.

C'est également au début du *linker script* que sont déclarées les tailles de la pile (*stack*) et de la zone d'allocation dynamique de mémoire (*heap*). Ces paramètres doivent être dimensionnés

selon les besoins du programme : s'ils sont trop faibles, les *malloc* ne se feront pas correctement. Pour le décodeur MPEG-2, le *linker script* prévoit **300 Ko** de pile et **200 Ko** pour les allocations dynamiques.

Lors de la compilation, le *linker script* est pris en compte de la manière suivante :
- sous EDK, la commande *Tools* > *Compiler Options* permet d'afficher et de modifier les options du compilateur. L'onglet *Directories* permet d'inclure, lors de la compilation, des fichiers et des dossiers autres que les contenu des répertoires *ppc405_0/lib/* et *ppc405_0/include/* qui sont inclus automatiquement. Il suffit de déclarer l'adresse du *linker script* pour que celui-ci soit pris en compte lors de la prochaine compilation.
- avec la console *xygwin*, la ligne de commande de compilation doit comporter les options `-Wl,-T -Wl,<adresse du linker script>` pour que le compilateur prenne en compte les données du *linker script*.

Deux *linker scripts* distincts sont utilisés dans le projet. Le premier déclare une **BRAM** sur le bus PLB, de **16 Ko**, et ne déclare pas la SDRAM, ceci afin d'éviter que l'exécutable tournant sur la BRAM (et dont le but est seulement le démarrage du processeur) ne mette des données sur la SDRAM et que les deux programmes ne se perturbent mutuellement. Le second *linker script*, destiné au décodeur proprement dit, déclare une **SDRAM de 16 Mo** environ au lieu des 32 Mo disponibles ; la mémoire restante est réservée aux fichiers. Pour la même raison que précédemment, ce *linker script* ne déclare pas de BRAM.

3.2.3 - Montage de la SDRAM

Sur la plate-forme Virtex-II Pro utilisée au laboratoire, la SDRAM peut en principe être montée soit sur le bus PLB, soit sur le bus OPB. Dans la pratique, il vaut mieux installer la SDRAM sur le bus **PLB** ; en effet, le bus OPB est séparé du processeur par une unité *plb2opbbridge* qui induit un temps de latence supplémentaire et provoque également certaines perturbations lors de l'exécution des programmes (accès trop long à certaines fonctions ...)

En montant la SDRAM sur le bus PLB, il faut vérifier certains paramètres, car EDK ne donne pas les bons paramètres lors de la création du projet. Il faut mettre à 1 le paramètre ***C_INCLUDE_BURST_CACHELN_SUPPORT*** ; ce paramètre permet d'inclure le support PLB ainsi que l'accès à un cache de transfert. Si ce paramètre n'est pas activé, il est impossible d'écrire dans la SDRAM sur le bus PLB.

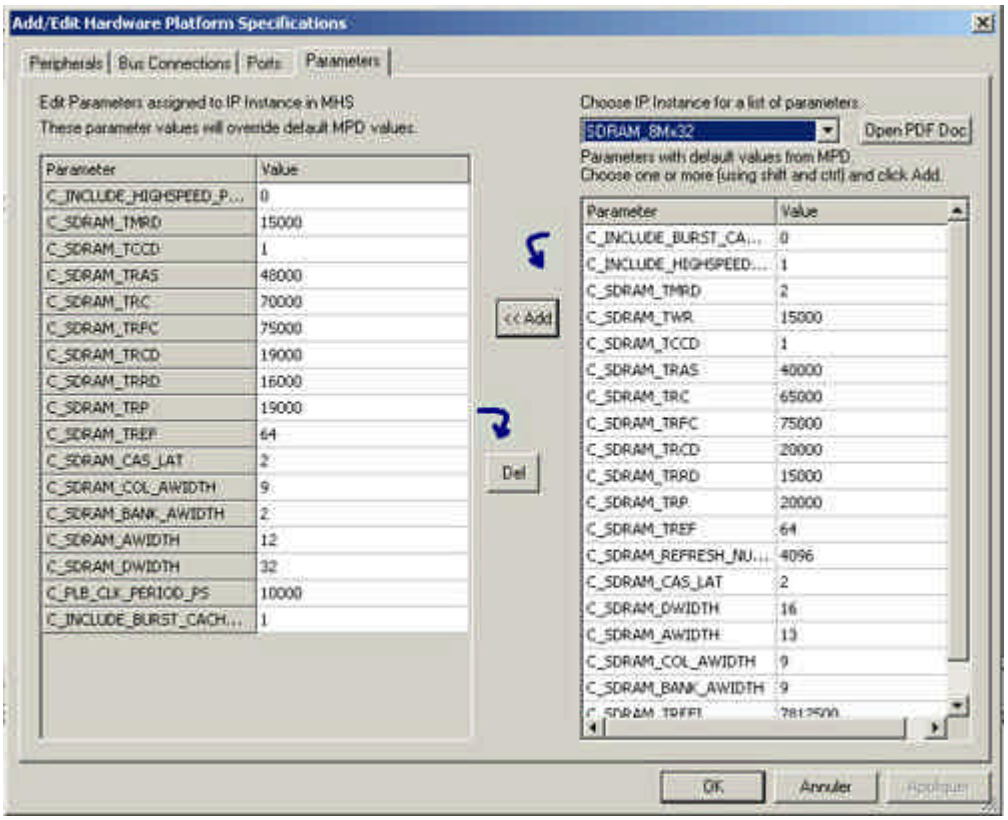


Figure 21 : Réglage des paramètres de la SDRAM

Dans le projet, l'adresse de base de la SDRAM sur le bus PLB est `0xfc000000`.

Lors du chargement de programmes test sur la SDRAM, un problème s'est posé : la fonction C `getchar`, indispensable au bon fonctionnement des fonctions de communication par *Xmodem* (*voir chapitre 3.3.4*), ne marchait pas. Elle a été remplacée par une fonction intervenant directement au niveau des registres de l'UARTlite. La mise en place de cette fonction, baptisée `recvchar`, a permis de gagner de la place mémoire en évitant d'utiliser des fonctions trop "lourdes". (*Voir l'annexe consacrée au protocole Xmodem pour le détail de la fonction.*)

3.3 - Adaptation du système de fichiers

3.3.1 - Le *Memory File System (MFS)*

Parallèlement à l'implémentation du programme proprement dit, il faut se poser la question du traitement des fichiers. En effet, le logiciel de décodage se base sur le **traitement d'un fichier source** (la séquence MPEG-2) et la restitution des images décodées sous forme de fichiers ; or, la plate-forme ne dispose pas de disque dur pour le stockage de ces fichiers. Cependant, il est possible de gérer un système de fichiers sur RAM en utilisant le *Memory File System* (*MFS*) qui permet le stockage d'une hiérarchie de fichiers sur une RAM, une ROM ou une mémoire Flash.

Le *MFS* se présente sous la forme **d'un ensemble de listes doublement chaînées de blocs-mémoire de 528 octets** (soit 512 octets de données) chacun. Il existe trois types de blocs distincts :

- *MFS_BLOCK_TYPE_FILE* : bloc appartenant à un fichier ;
- *MFS_BLOCK_TYPE_DIR* : bloc appartenant à un répertoire ;
- *MFS_BLOCK_TYPE_EMPTY* : bloc vide.

Chaque fichier du *MFS* se présente comme une **liste doublement chaînée de blocs** (les dossiers ne sont pas utilisés dans le projet, ils ne seront donc pas abordés ici). Le premier bloc du fichier est une référence au répertoire auquel il appartient ; les blocs suivants contiennent les données du fichier proprement dit. Les blocs vides forment une entité à part, la *free list*, qui est la liste doublement chaînée de tous les blocs vides. Lors d'une écriture dans un fichier, si on a besoin d'ajouter un nouveau bloc, on déchaîne de la *free list* son premier bloc et on le rechaîne au fichier qui l'utilise.

3.3.2 - Utilisation du *MFS*

Pour utiliser le *MFS*, il faut tout d'abord incorporer dans le projet la bibliothèque de gestion de ce système, *LibXil MFS*, constituée des sources `mfs_filesys.c` (fonctions d'accès aux fichiers et répertoires du MFS) et `mfs_filesys_util.c` (utilitaires *MFS* additionnels) (la manière d'incorporer cette bibliothèque est indiquée dans *l'annexe consacrée à EDK*) et inclure, dans les parties du programme utilisant les accès aux fichiers, le fichier d'en-tête `xilmfs.h`.

Ces parties du programme doivent subir une légère modification. En effet, les fonctions d'accès aux fichiers du *MFS* sont légèrement différentes. Il faut donc modifier leur nom mais aussi, pour certaines, leurs paramètres :

```
open(filename,O_RDONLY)
-> mfs_file_open(filename,MFS_MODE_READ)
open(filename,O_WRONLY)
-> mfs_file_open(filename,MFS_MODE_WRITE)
close(fd)
-> mfs_file_close(fd)
read(fd,buffer,buflen)
-> mfs_file_read(fd,buffer,buflen)
write(fd,buffer,buflen)
-> mfs_file_write(fd,buffer,buflen)
```

Afin de pouvoir utiliser le système de fichiers, celui-ci doit être initialisé au début du programme. Il existe une fonction d'initialisation du *MFS* :

```
void mfs_init_fs(int numbytes, char *addr, int MODE)
```

où *numbytes* représente le nombre d'octets alloués pour le système, *addr* l'adresse de début (base) du système et *MODE* le mode d'initialisation du *MFS* :
MFSINIT_NEW pour un nouveau système (vide) ;
MFSINIT_IMAGE pour reprendre un système pré-téléchargé dans la mémoire ;
MFSINIT_ROM_IMAGE comme pour le mode précédent mais en lecture seule.

3.3.3 - Les limites du *MFS*

La solution retenue au départ était d'utiliser le second mode, *MFSINIT_IMAGE*, avec un système de fichiers préalablement généré sur le PC hôte.

En effet, la fonction `xdownload` du *XMD* permet de télécharger des données vers la plate-forme à une adresse donnée par l'utilisateur (celle de la SDRAM par exemple). De plus, Xilinx fournit un programme nommé *mfsgen* qui permet de générer un système de fichiers sur le PC hôte à partir de fichiers et/ou de répertoires existants. La première solution retenue consistait donc à générer un système de fichiers, nommé *test.mfs*, à partir du fichier source MPEG-2, puis à le télécharger sur la plate-forme à l'adresse prévue pour le *MFS*.

Malheureusement, cette solution s'est révélée inefficace ; en effet, un bug dont l'origine n'a pas été trouvée perturbe le téléchargement du *MFS* qui arrive sur la plate-forme corrompu (*free list* vide, chaînage des blocs de fichier perturbé ...) et donc **inutilisable par le décodeur** (celui-ci parvient quand même à lire le fichier mais renvoie des messages d'erreur et des images résultats erronées).

3.3.4 - Transfert des fichiers vers la plate-forme

En revanche, une autre solution a pu être retenue : le logiciel HyperTerminal est capable d'envoyer et de recevoir des fichiers **par la liaison série**, en utilisant différents protocoles de transfert de fichiers. Il a donc été décidé d'envoyer la séquence MPEG-2 au décodeur au début du programme, et de recevoir les images résultantes à la fin du programme, par la liaison série. Le protocole choisi pour ces transferts est le protocole *Xmodem*, en raison de sa simplicité de mise en oeuvre *(voir l'annexe consacré à ce protocole pour plus de détails)*.

Le *MFS* de la plate-forme est donc initialisé avec le paramètre *MFSINIT_NEW*. En raison de la taille importante des fichiers résultats, le *MFS* a été dimensionné à 500 blocs mémoire, soit environ 256 Ko.

En raison d'une limitation de la fonction `xmodem_recoit_fichier`, il faut démarrer l'envoi du fichier MPEG-2 par HyperTerminal avant de lancer le programme de décodage : en effet, le décodeur n'envoie qu'une fois le signal qui commande le début du transfert.

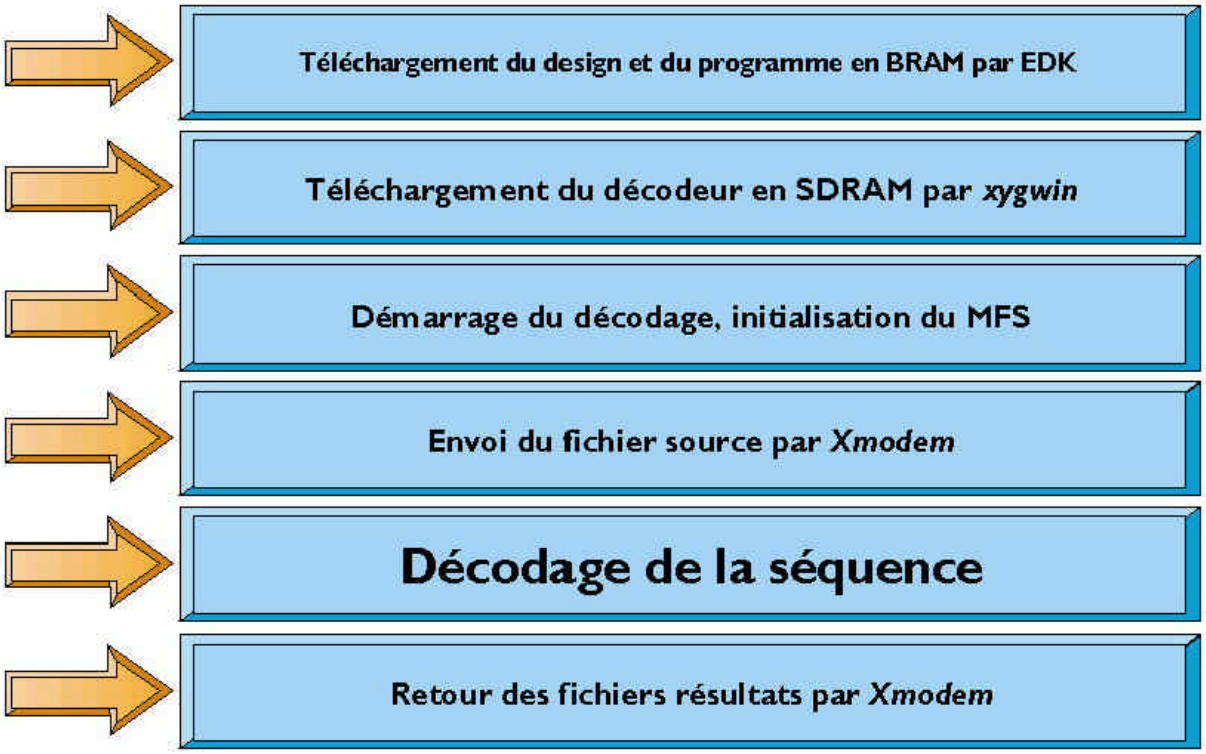


Figure 22 : Etapes du décodage MPEG-2 sur plate-forme

Une fois les fichiers résultats transférés par la plate-forme, il ne reste plus qu'à les ouvrir pour vérifier leur adéquation à la séquence initiale. Pour cela, il a été nécessaire de télécharger une visionneuse adaptée, *XnView*, car les formats des fichiers renvoyés par le décodeur ne sont pas reconnus par les outils Windows standard.

Le décodeur prévoit plusieurs formats disponibles ; ceux qui sont lisibles par *XnView* sont au nombre de 2 : **Portable Pix Map (.ppm)** et **Targa TrueVision (.tga)**.

3.4 - Résultats de l'implémentation

3.4.1 - Vérification des résultats

Afin de vérifier le résultat du décodage par le programme sur la plate-forme, et pour s'assurer de l'intégrité du décodeur même après la désactivation des fonctions se rapportant aux profils *scalable*, les modifications apportées au programme (à l'exception de celles ayant trait au *MFS*) ont également été faites à des sources compilées sur le PC hôte par le compilateur classique *gcc*. La comparaison des images renvoyées par ce décodeur "témoin" et par le programme de la plate-forme permet de vérifier la pertinence du décodage de ce dernier.

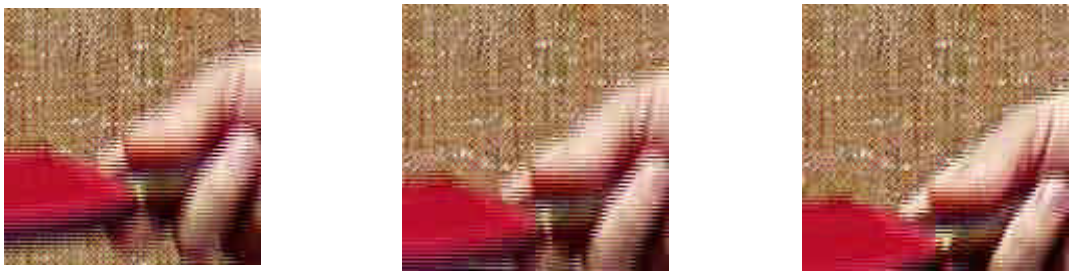


Figure 23 : Résultats du décodeur "témoin"

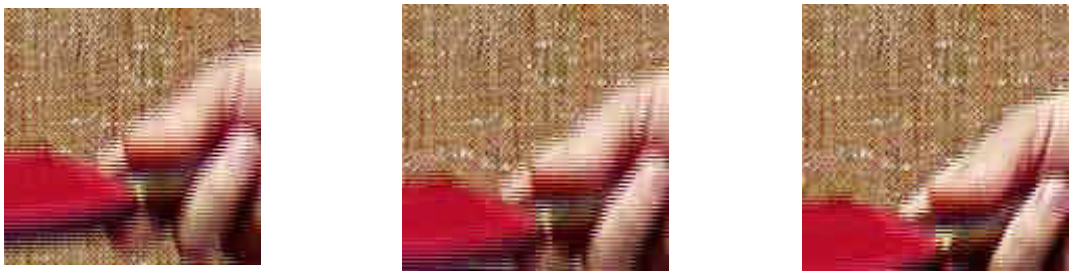


Figure 24 : Résultats du décodeur sur plate-forme

La comparaison de ces images montre que le décodeur implémenté sur la plate-forme Virtex-II Pro renvoie les mêmes images que le décodeur MPEG-2 original. L'implémentation du décodeur logiciel sur le Power-PC est donc réussie.

3.4.2 - Les futures améliorations

Jusque-là, seule l'implémentation logicielle sur le Power-PC a été réalisée ; il n'y a aucune transformation des fonctions logicielles en blocs hardware. La conception de la fonction de déquantification étant presque terminée, cette fonction sera prochainement implémentée sur le FPGA et reliée au Power-PC afin d'obtenir un décodeur MPEG-2 possédant une véritable architecture mixte ; elle sera suivie de la fonction d'IDCT.

Il faudra également évaluer avec précision les performances du décodeur et vérifier qu'il est bien en mesure d'effectuer un décodage en temps réel des images de la séquence vidéo.

Enfin, une autre amélioration de ce projet consisterait en l'intégration d'un système complet de décodage MPEG-2 sur la plate-forme, où une interface de type *RocketIO* (interface E/S série à très haute vitesse) recevrait le flux MPEG-2 et où les images résultantes seraient traitées par une interface DVI puis sorties, par l'intermédiaire d'autres interfaces *RocketIO*, sur un écran LCD, permettant ainsi une visualisation d'une séquence MPEG-2 en temps réel.

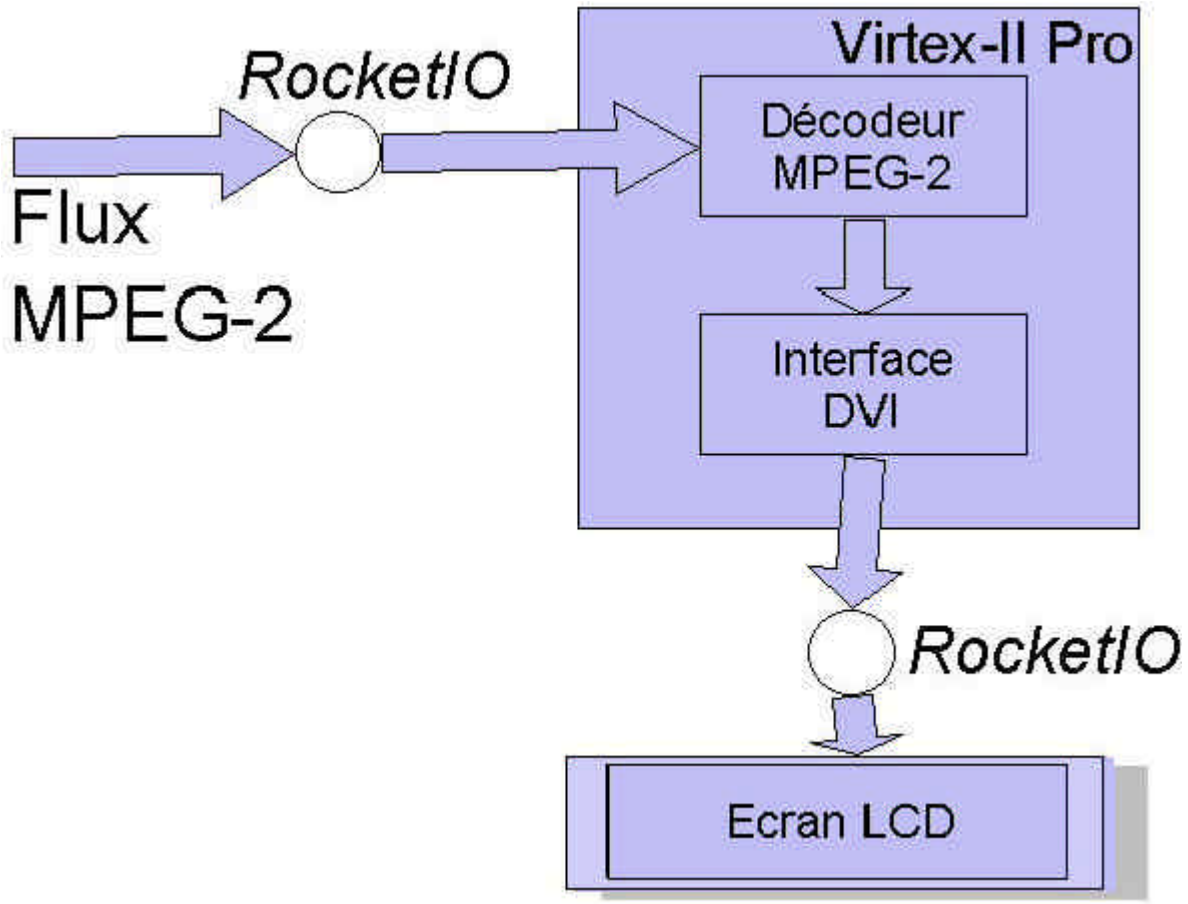


Figure 25 : Schéma simplifié du futur décodeur MPEG-2 complet

3.5 - Conclusion

Ce stage m'a permis d'effectuer une approche pratique de la recherche, mais aussi du développement de logiciel embarqué et de la conception de fonctions sur FPGA.

J'ai appris à concevoir des fonctions hardware dans le but de réduire le temps de calcul et d'améliorer les performances d'un système, à effectuer des optimisations de structures fonctionnelles (en particulier la parallélisation) et à évaluer les cadences de calcul des fonctions avant et après optimisation.

J'ai pu effectuer une approche très concrète de la programmation sur une plate-forme de développement mixte complexe telle que le Virtex-II Pro, de l'utilisation d'outils de développement avancés, ainsi que de la programmation sur des processeurs et des FPGA évolués.

J'ai eu l'occasion de traiter des problèmes concrets de déboguage, en particulier lors de l'implémentation du système de fichiers sur la plate-forme. J'ai dû également rechercher des solutions pratiques selon les possibilités offertes par la plate-forme, et les adapter à la situation.

J'ai pu, enfin, appréhender la norme de compression vidéo MPEG-2 et avoir un aperçu des techniques de compression utilisées en multimédia, y compris pour des normes plus complexes telles que le MPEG-4, ainsi que les formats de compression audio.

Bibliographie

Ouvrages :

H. BENOÎT - *La Télévision numérique : MPEG-1, MPEG-2 & système européen DVB (2ème édition)* - Editions Dunod, 1998 - pp. 39-63

M. BARLAUD / C. LABIT - *Compression et codage des images et des vidéos* - Hermès Sciences Publications, 2002 - pp. 113-120, 188-192

Articles de recherche :

M. SIMA, S. COTOFANA, J. VAN EIJNDHOVEN, S. VASSILIADIS, K. VISSERS - *An 8x8 IDCT implementation on an FPGA-Augmented TriMedia* - FFCM 2001, Rohnert Park, California

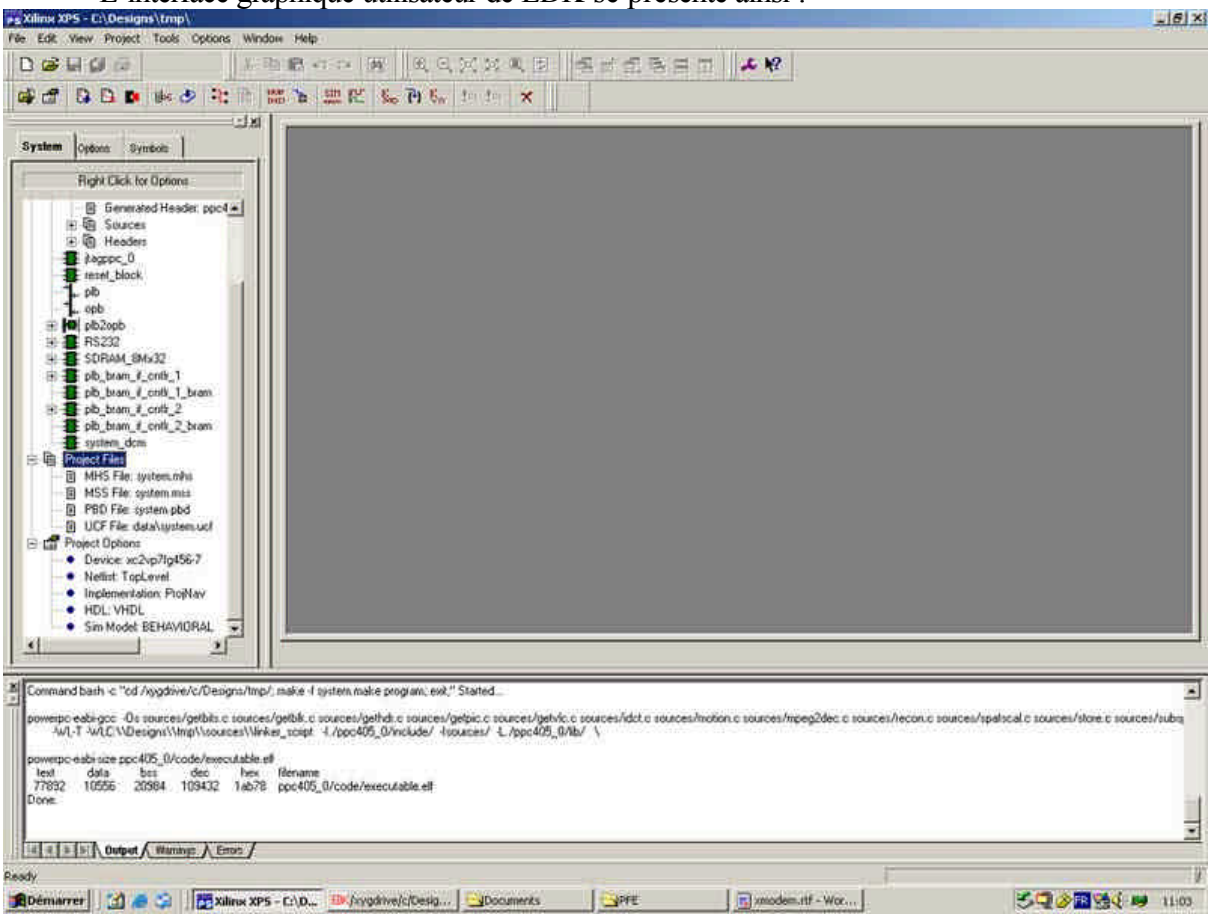
K. BUKHARI, G. KUZMANOV, S. VASSILIADIS - *DCT and IDCT implementations on different FPGA technologies* - Delft University of Technology, 2002

Annexe : Xilinx Embedded Development Kit (EDK)

Introduction

Xilinx Embedded Development Kit (EDK) est un logiciel de développement compatible avec de nombreuses plate-formes mixtes vendues par Xilinx. Cet environnement de développement permet de configurer la plate-forme en sélectionnant ses éléments puis en compilant et en débuggant les programmes que l'on veut y faire tourner.

L'interface graphique utilisateur de EDK se présente ainsi :



Pour créer un projet sur plate-forme, EDK fournit un assistant de création, le *Base System Builder Wizard* qui permet de sélectionner rapidement le type de plate-forme utilisée et les éléments que l'on veut y intégrer. Par exemple, dans le cas du Virtex-II Pro utilisé au laboratoire ETIS, on sélectionne les options suivantes :

Le *Base System Builder Wizard* indique alors quels sont les éléments qui peuvent être sélectionnés : dans le cas du Virtex-II Pro, on a :

- le processeur Power-PC ;
- une interface série RS-232 de type UART (il y en a plusieurs types disponibles) ;
- quatre LEDs ;
- quatre boutons-poussoirs ;
- un interrupteur DIP 8 bits ;
- une SDRAM de 32 Mo ;
- des RAMs ;
- des *timers*.

Mise en place de la partie matérielle

Une fois que les éléments de la plate-forme ont été sélectionnés (et que d'autres ont été éventuellement ajoutés), on génère la *netlist* du projet grâce à la commande *Tools > Generate Netlist* de EDK. Après la génération de cette *netlist*, on peut exporter le projet vers Xilinx ISE Project Navigator par la commande *Tools > Export to Projnav*. EDK crée alors un fichier nommé system.npl qui sera utilisé par Project Navigator pour créer le fichier de configuration matérielle de la plate-forme. Une fois que ce fichier de configuration sera généré *(voir l'annexe sur ISE Project Navigator)* , la commande de EDK *Tools > Import from ProjNav* permet de le récupérer.

Note : pour exporter le projet vers Project Navigator, il faut indiquer à EDK qu'on choisit Project Navigator pour la synthèse de la partie matérielle ; pour cela, il faut l'indiquer dans les options du projet en cliquant sur *Options* > *Project Options*.

Mise en place de la partie logicielle

Quand la partie matérielle est définie, on peut incorporer des codes sources dans le processeur, puis les compiler afin de les télécharger sur la plate-forme. EDK supporte les sources C (fichiers .c) et leurs fichiers d'en-têtes (fichiers .h) mais aussi les sources C++ (fichiers .cpp ou .c++).

Lors de la sélection des éléments à incorporer, EDK génère automatiquement la liste des pilotes à utiliser : ceux-ci sont listés dans un fichier nommé system.mhs. Si on veut utiliser des bibliothèques logicielles supplémentaires, comme LibXil File ou LibXil MFS, c'est dans ce fichier qu'elles doivent être déclarées :

```
BEGIN LIBRARY
parameter LIBRARY_NAME = xilmfs
parameter LIBRARY_VER = 1.00.a
parameter numbytes = (le nombre d'octets prévus pour le
système de fichiers)
parameter base_address = (l'adresse de base du système de
fichiers, dans la SDRAM de préférence)
parameter init_type = MFSINIT_IMAGE (image d'un système
généré par mfsgen)
parameter need_utils = (utilisation de certaines fonctions
utilitaires, true/false)
END
```

EDK génère les bibliothèques logicielles nécessaires au projet en se basant sur ce fichier, lorsqu'on lance la commande *Tools* > *Generate Libraries*. Ces bibliothèques sont copiées à partir des données de EDK et compilées dans les répertoires `ppc405_0/lib` et `ppc405_0/include`. Une fois cette commande terminée, on peut compiler les sources ajoutées au projet par la commande *Tools* > *Compile Program Sources*, après avoir réglé les options du compilateur en cliquant sur *Options* > *Compiler Options* :

- choix du compilateur et de l'archiveur utilisés : pour un Power-PC il s'agit respectivement de *powerpc-eabi-gcc* et de *powerpc-eabi-ar* ;
- choix des bibliothèques et des fichiers à inclure lors de la compilation : les contenus des deux répertoires générés par *Generate Libraries* (*ppc405_0/lib/* et *ppc405_0/include/*) sont automatiquement inclus ;
- niveau d'optimisation de la compilation : permet une réduction du volume de l'exécutable final ;
- choix de l'OS utilisé par le processeur, celui-ci peut utiliser l'OS temps réel VxWorks ou (dans le cas du projet) tourner en *standalone* sans OS ;
- utilisation des symboles de debug ;
- choix du *linker script* utilisé ;
- ajout d'autres options de compilation (*flags* supplémentaires par exemple).

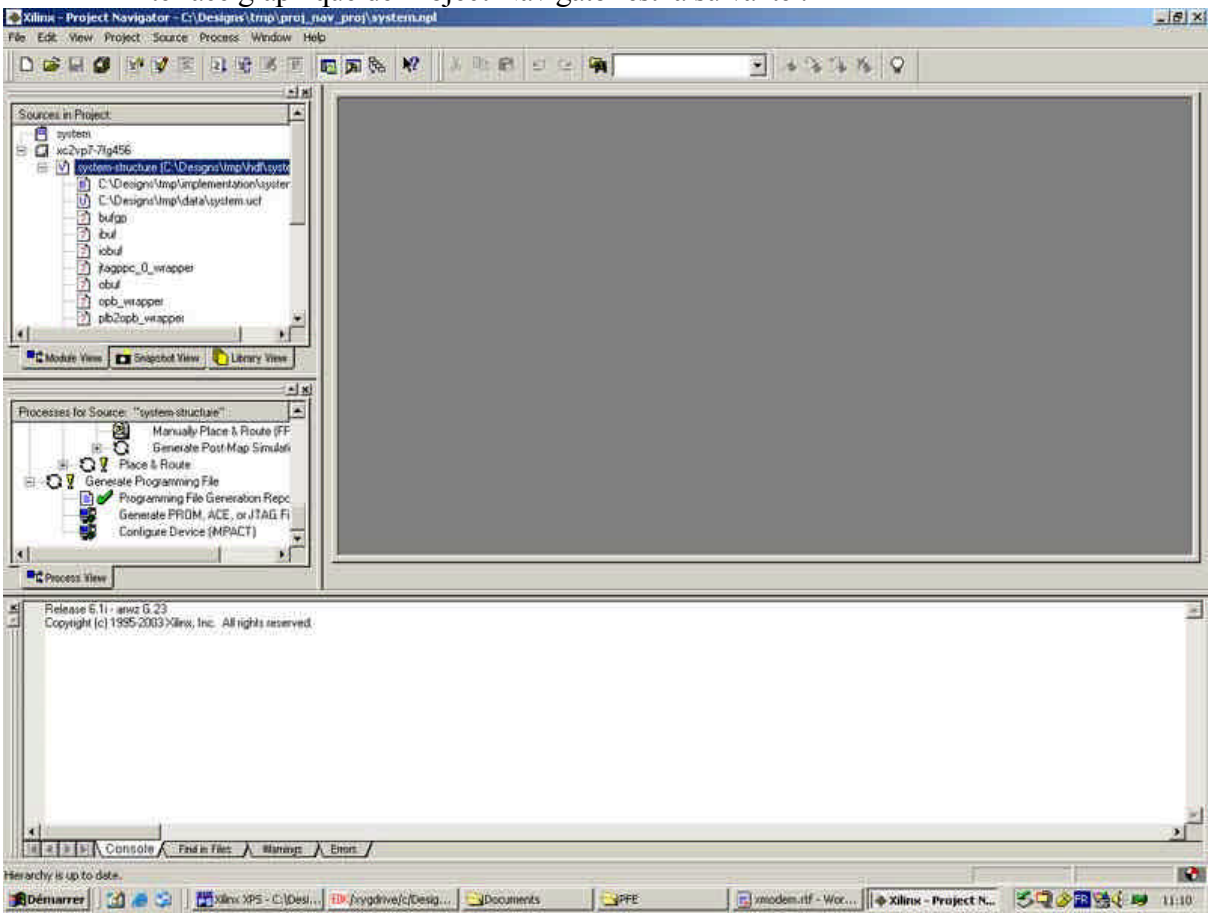
Téléchargement du programme

Lorsque la compilation est terminée correctement, la partie logicielle est prête. Il suffit alors de réunir les deux parties du projet par la commande *Tools > Update Bitstream* ; on peut alors effectuer le téléchargement du fichier sur la plate-forme avec *Tools > Download*. EDK télécharge alors le projet sur la plate-forme par l'intermédiaire du câble JTAG.

Annexe : Xilinx ISE Project Navigator

Xilinx ISE Project Navigator est un logiciel de synthèse hardware des projets destinés aux plate-formes FPGA ou mixtes de Xilinx. Dans le cadre du projet sur la plate-forme Virtex-II Pro de Xilinx, il est utilisé pour la synthèse de la partie matérielle du projet.

L'interface graphique de Project Navigator est la suivante :



Synthèse matérielle d'un projet

Une fois que EDK a créé par la commande *Export to ProjNav* le fichier .npl *(voir l'annexe sur Embedded Development Kit)*, on peut l'ouvrir avec ISE. EDK a créé le design matériel sous la forme d'un fichier VHDL et de quelques autres sources contenant des données complémentaires sur l'implémentation.

En sélectionnant le fichier VHDL du projet, on peut visualiser les différents processus qui lui sont applicables. Pour générer le fichier de configuration de la plate-forme, il faut passer par plusieurs étapes représentées chacune par un processus différent :

- *Synthesize* effectue la synthèse des fichiers VHDL ;
- *Implement Design* vérifie que le design est compatible avec les ressources matérielles réelles de la plate-forme et effectue le placement et le routage ;
- *Generate Programming File* convertit les données issues des processus précédents en un fichier de configuration portant l'extension .bit.

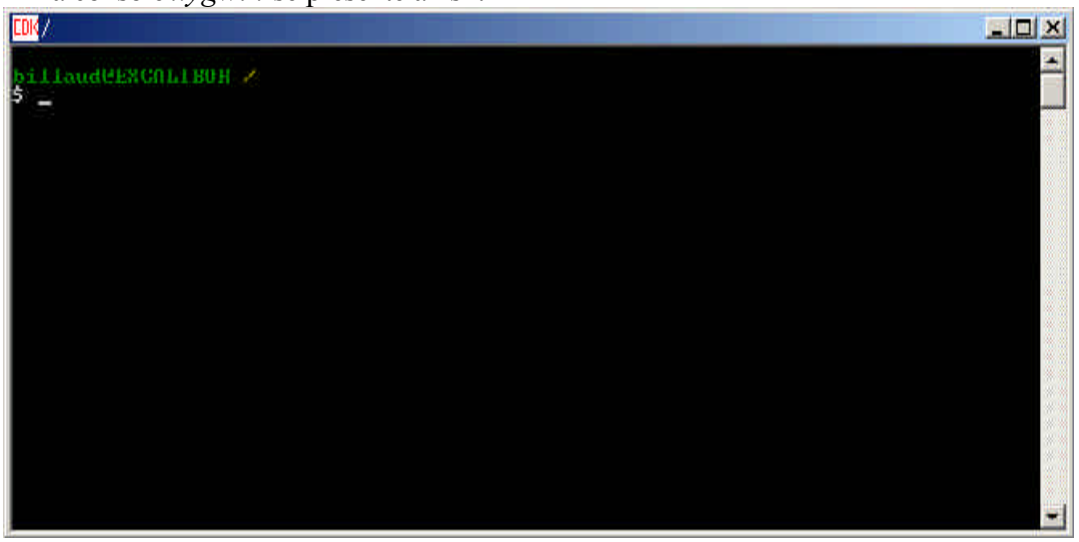
Note : si le matériel utilisé est un FPGA simple ou si le processeur n'est pas utilisé, ISE Project Navigator permet de télécharger un design hardware seul sur la plate-forme. Il utilise pour cela le logiciel de téléchargement iMPACT ainsi que le câble JTAG.

Annexe : XMD et la console *xygwin*

Le *Xilinx Microprocessor Debugger (XMD)* est un programme de déboguage pour les processeurs des plate-formes Xilinx. Il est prévu pour fonctionner aussi bien avec un processeur MicroBlaze ou, dans le cas du projet, avec un Power-PC.

Le XMD est appelé par la console *xygwin* fourni avec Xilinx EDK. Cette console est une variante de la console *Cygwin* de Cygnus, qui est un émulateur de commandes Unix sous Windows. Cette console permet donc non seulement d'appeler XMD mais aussi d'appeler le compilateur *powerpc-eabi-gcc* pour la compilation de sources en-dehors d'EDK (par exemple pour compiler les sources destinées à la SDRAM) ainsi que d'autres commandes dont on pourrait avoir besoin, entre autres le désassemblage de l'exécutable par la commande `powerpc-eabi-objdump -D executable.elf > objdump`, qui crée un fichier *objdump* contenant le programme désassemblé.

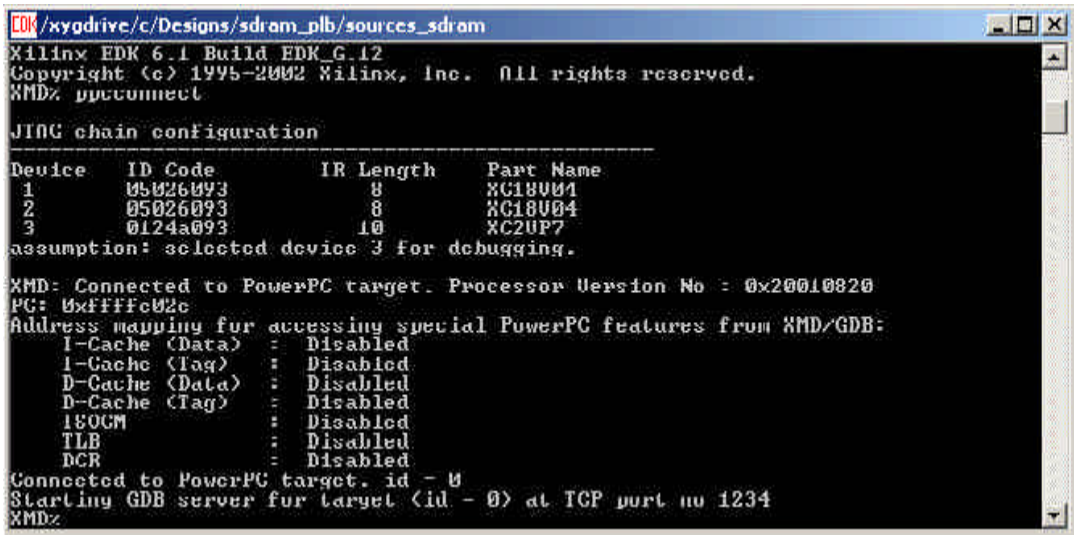
La console *xygwin* se présente ainsi :



Pour lancer XMD et se connecter au Power-PC il faut taper les commandes suivantes :

```
xmd
ppcconnect
```

On est alors dans l'interface de XMD.



Remarque importante : Lorsque XMD est connecté au Power-PC, il monopolise le câble de liaison JTAG ; tout téléchargement de design par EDK ou Project Navigator est alors impossible. Pour utiliser à nouveau le téléchargement d'EDK ou de Project Navigator, il faut d'abord quitter XMD par la commande `exit`.

XMD admet un grand nombre de commandes ; les plus utiles dans le cadre du téléchargement d'un programme en SDRAM sont :

- `xdownload` : cette commande permet de télécharger des données ou des programmes sur un processeur cible ; pour la plate-forme utilisée, il n'y a qu'une cible possible et elle est toujours désignée par le nombre 0. La commande `xdownload` s'utilise de deux façons différentes :

`xdownload 0 -data test.mfs 0xfd100000` : télécharge les données du fichier *test.mfs* dans la mémoire du processeur cible à l'adresse de base indiquée ;

`xdownload 0 executable.elf` : télécharge le programme *executable.elf* dans la mémoire du processeur cible. Il est inutile de préciser une adresse car celle-ci est fournie dans les en-têtes du programme. Lors du téléchargement d'un programme, le pointeur de commande (PC) du processeur cible est automatiquement changé pour l'adresse de la zone de boot du programme.

- `xcontinue` : continue l'exécution d'un programme par le processeur cible, à partir de l'adresse indiquée ou, si aucune adresse n'est indiquée, de la position actuelle du PC. Cette commande s'utilise de la manière suivante :

```
xcontinue 0 <adresse éventuelle>
```

- `xrmem` : outil très utile pour la vérification du téléchargement, cette commande lit la mémoire du processeur cible à partir de l'adresse indiquée. Elle renvoie le nombre d'octets désirés (ou 1 octet si aucun nombre n'est précisé) sous forme décimale. Elle s'utilise de la manière suivante :

```
xrmem 0 <adresse de base> <nombre d'octets éventuel>
```

La comparaison entre une lecture par `xrmem` et un fichier *objdump* permet de vérifier si le téléchargement d'un programme s'est bien passé.

- `xwmem` : pendant de `xrmem`, cette commande permet d'écrire un octet dans la mémoire du processeur cible à l'adresse indiquée. Elle s'utilise de la manière suivante :

```
xwmem 0 <adresse> <valeur de l'octet>
```

- `xbreakpoint` : permet de placer un point d'arrêt à l'adresse mémoire indiquée. Cette commande s'utilise de la manière suivante :

```
xbreakpoint 0 <adresse du point d'arrêt> sw
```

- `xremove` : enlève un point d'arrêt à l'adresse indiquée. À noter que la remise à zéro du processeur efface automatiquement tous les points d'arrêt.

Annexe : le protocole *Xmodem*

Le protocole *Xmodem* est un protocole simple de transfert de fichiers. Il ne transmet pas de nom de fichier mais uniquement son contenu, sous forme de paquets de 128 octets de données auxquels il faut ajouter les en-têtes et la somme de contrôle, soit 132 octets en tout. Un paquet *Xmodem* se présente de la manière suivante :

<SOH> <N° paquet> <255-n° paquet><128 octets de données> <octet de contrôle>

SOH *(Start Of Header)* est le code de départ du paquet *Xmodem*, il est égal à 0x01.

L'octet de contrôle est le dernier octet de la somme des données précédentes (SOH + numéros + données), il est utilisé pour le contrôle des erreurs. Si cet octet ne correspond pas à la somme des données, le récepteur envoie un signal NAK *(Negative AcKnowledgement)* égal à 0x15 et le paquet est réenvoyé. Si l'envoi s'est passé correctement, le récepteur renvoie un signal ACK *(ACKnowledgement)* égal à 0x06 et le transfert peut continuer par le paquet suivant.

Le début du transfert ne donne pas lieu à l'émission d'un signal particulier par l'émetteur. C'est le récepteur, en se plaçant en attente d'un fichier, qui envoie régulièrement des signaux NAK à l'émetteur. Sur un de ces signaux NAK, l'émetteur transmet le premier paquet du fichier.

La fin du transfert se matérialise par l'émission d'un octet de fin de transmission par l'émetteur. Cet octet, EOT *(End Of Transmission)*, est égal à 0x04. Le dernier paquet du fichier transmis contient la fin de celui-ci et un bourrage d'octets nuls.

HyperTerminal est capable de recevoir des fichiers par le protocole *Xmodem* : il suffit de lui indiquer le chemin où placer le fichier reçu, ainsi que son nom (puisque *Xmodem* ne transmet pas le nom du fichier). L'envoi des fichiers de la plate-forme vers l'HyperTerminal du PC hôte se fait par la liaison série à 115200 bauds, grâce aux fonctions du fichier `xmodem.c`.

De la même façon, HyperTerminal peut envoyer des fichiers par le protocole *Xmodem* : il suffit de lui indiquer l'emplacement du fichier à transmettre. Il attend alors l'envoi d'un NAK du récepteur pour démarrer le transfert.

Note : Il existe deux types de protocoles *Xmodem* différents. Celui utilisé ici utilise une somme arithmétique sur 8 bits et le signal de départ est un NAK. Il existe un autre type de protocole où la somme de contrôle est une somme dite "CRC" basée sur un Ou exclusif et où le signal de départ correspond au caractère 'C'. Ce mode, plus complexe, n'a pas été implémenté ici.

Code source du fichier `xmodem.c` contenant les fonctions d'émission Xmodem :

```
/* xmodem.c - Gestion du transfert des fichiers résultats par le protocole
Xmodem */

/* Ce fichier est prévu pour gérer des paquets de 128 octets */

#include "xparameters.h"
#include <stdio.h>
#include <fcntl.h>
#include "xilmfs.h"

#define SOH 0x01
#define NAK 0x15
#define ACK 0x06
```

```
#define EOT 0x04

void xmodem_ouvre(void);
void xmodem_cree_paquet(int fsource, char *paquet, char num_paquet, char
*fin);
void xmodem_envoie_paquet(char *paquet);
void xmodem_ferme(void);

void xmodem_r_ouvre(void);
char xmodem_recoit_paquet(int fic, char *num_paquet, char *fin);


/* fonction de réception d'un caractère directement par les registres de
l'UART
(remplace getchar() qui ne marche pas) */

char recvchar(void)
{
      char c;
      volatile int *addr=(int*)XPAR_RS232_BASEADDR;

      while((*(addr+2)&1)!=1)
      {
      }
      c=(char)*addr;
      return c;
}


/* Emission d'un fichier par XModem */
void xmodem_emet_fichier(char *fichier)
{
      char paquet[132]; /* 128 bits de données + en-têtes + contrôle
d'erreur */
      char num_paquet;
      volatile char rep;
      char fin;
      int fsource;
      int error;


      fin = 0;

      if((fsource=mfs_file_open(fichier,MFS_MODE_READ))<0) /* ouverture du
fichier */
      {
            xil_printf("Fichier non ouvert\r\n");
            return;
      }
      xmodem_ouvre();
      num_paquet=0x01;
      while(fin!=1)
      {
            xmodem_cree_paquet(fsource,paquet,num_paquet,&fin);
            xmodem_envoie_paquet(paquet);
            rep=recvchar();
            while(rep!=ACK)
            {
```

```
                        xmodem_envoie_paquet(paquet);
                        rep=recvchar();
                }
                num_paquet++;
         }
        xmodem_ferme();
        mfs_file_close(fsource); /* fermeture du fichier */
        return;
}

/* Ouverture de la connexion XModem en émission */
void xmodem_ouvre(void)
{
        char debut;

        debut=0;
        while(debut!=NAK) /* attente du NAK de départ */
        {
                debut=recvchar();
        }
        return;
}

/* Création d'un paquet de 128 octets à partir du fichier */
void xmodem_cree_paquet(int fsource, char *paquet, char num_paquet, char
*fin)
{
        int somme;
        int i;
        char buffer;
        int cursor;

        buffer=0x00; /* initialisation */
        for(i=0;i<131;i++)
                paquet[i]=0x00;

        paquet[0]=(char)SOH;
        paquet[1]=num_paquet;
        paquet[2]=(255-num_paquet);

        /* cursor=(128*(num_paquet-1));
        mfs_file_lseek(fsource,cursor,MFS_SEEK_SET); */

        for(i=3;i<=130;i++)
        {
                if(mfs_file_read(fsource,&buffer,1)==0)
                {
                        i=131;
                        *fin=1;
                }
                else
                {
                        paquet[i]=buffer;
                }
        }
        somme=0;
        for(i=0;i<=130;i++)
                somme=somme+paquet[i];
```

```
	somme=(char)somme;
	paquet[131]=somme;

	return;
}

/* Emission d'un paquet de données */
void xmodem_envoie_paquet(char *paquet)
{
	int i;

	for(i=0;i<=131;i++)
		xil_printf("%c",paquet[i]);
	return;
}

/* Fin de communication XModem */
void xmodem_ferme()
{
	volatile char rep;

	xil_printf("%c",EOT);
	rep=0;
	while(rep!=ACK)
		{
			rep=recvchar();
		}
	return;
}


/* Réception d'un fichier Xmodem */
void xmodem_recoit_fichier(char *fichier) /*158*/
{
	int fic;
	char fin;
	char num_paquet;
	char rep;

	num_paquet=0x01;
	xmodem_r_ouvre();
	if((fic=mfs_file_open(fichier,MFS_MODE_CREATE))<0)
	{
		return;
	}
	while(fin!=1)
	{
		rep=xmodem_recoit_paquet(fic,&num_paquet,&fin);
		xil_printf("%c",rep);
		/* réception du paquet et envoi de la réponse */
	}
	mfs_file_close(fic);
	return;
}

/* Ouverture de connexion Xmodem en réception */
void xmodem_r_ouvre(void)
{
```

```
	xil_printf("%c",NAK);
}

/* Réception d'un paquet Xmodem */
char xmodem_recoit_paquet(int fic, char *num_paquet, char *fin)
{
	char paquet[132];
	int i;
	char rep;
	char somme;

	paquet[0]=recvchar();
	if(paquet[0]==EOT) /* fin de transmission : on renvoie seulement ACK
et on termine la procédure */
	{
		*fin=1;
		rep=ACK;
	}/*200*/
	else
	{
		for(i=1;i<=131;i++)
			paquet[i]=recvchar();

		for(i=0;i<=130;i++)
			somme=somme+paquet[i];

		somme=(char)somme;


if((*num_paquet!=paquet[1])||((255-(*num_paquet))!=paquet[2])||(somme!=paqu
et[131]))
		{
			rep=NAK;
		}
		else
		{
			rep=ACK;
			mfs_file_write(fic,paquet+3,128); /* paquet OK : écriture
dans le fichier cible */
			(*num_paquet)++; /* prêt pour le paquet suivant */
		}
	}
	return rep;
}
```